Porting and Optimizing HPC Applications for Arm SVE

Version 1.1.0

Porting and Optimizing HPC Applications for Arm SVE

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	17 May 2019	Non-Confidential	First release.
0110-00	05 June 2019	Non-Confidential	Document update to version 1.1.

Contents

Porting and Optimizing HPC Applications for Arm SVE

List of Figures
Porting and Optimizing HPC Applications for Arm SVE

List of Tables
Porting and Optimizing HPC Applications for Arm SVE

Preface

This preface introduces the *Porting and Optimizing HPC Applications for Arm SVE* .

It contains the following:
- *About this book* on page 9.

About this book

Using this book

This book is organized into the following chapters:

Chapter 1 Porting to Arm Resources
This guide supplements the other resources Arm provides to help you get started with porting your applications to Arm, with a focus on optimizing for the Arm Scalar Vector Extension (SVE). This chapter provides more information about additional porting resources available from Arm.

Chapter 2 Explore the Scalable Vector Extension (SVE)
Scalable Vector Extension (SVE) is an optional vector extension for AArch64, introduced in Armv8.2-A. Unlike other SIMD architectures, SVE does not define the size of the vector registers, but constrains it to a range of possible values, from a minimum of 128 bits up to a maximum of 2048 in 128-bit wide units. The CPU designer can implement the extension by choosing the vector register size that best suits the workloads the CPU is targeting. The design of SVE guarantees that the same program can run on different implementations of the instruction set architecture without the need to recompile the code.

Chapter 3 SVE Vector Length Agnostic programming
This chapter introduces the concept of Vector Length Agnostic (VLA) programming and provides SVE programming tips supported by examples in assembly that use the Arm C Language Extensions (ACLE) for SVE.

Chapter 4 Generic Vector and Matrix Operations Examples
This chapter discusses generic vector and matrix operations examples.

Chapter 5 Arm Instruction Emulator
Arm Instruction Emulator (ArmIE) runs on AArch64 platforms and emulates SVE instructions. ArmIE enables you to compile SVE code with Arm Compiler and run the SVE binary without SVE-enabled hardware. Based on the DynamoRIO dynamic binary instrumentation framework, ArmIE enables the customized instrumentation of SVE binaries allowing you to analyze specific aspects of runtime behavior.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (with "mono" underlined)
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Porting and Optimizing HPC Applications for Arm SVE* .
- The number 101726_0110_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Porting to Arm Resources

This guide supplements the other resources Arm provides to help you get started with porting your applications to Arm, with a focus on optimizing for the Arm Scalar Vector Extension (SVE). This chapter provides more information about additional porting resources available from Arm.

It contains the following section:
- *1.1 Resources* on page 1-12.

1.1 Resources

This guide supplements the other resources Arm provides to help you get started with porting your applications to Arm, with a focus on optimizing for the Arm Scalar Vector Extension (SVE). Other porting resources include:

- *Porting and Optimizing HPC Applications for Arm* guide, which includes information on migrating from other compilers to Arm Compiler
- *Application porting and tuning recipes*.
- Community-driven application porting recipes on the *Arm GitLab repository*
- *White papers*.
- *Arm Allinea Studio tool documentation*
- Blogs and forums on *Arm HPC Community*

Chapter 2
Explore the Scalable Vector Extension (SVE)

Scalable Vector Extension (SVE) is an optional vector extension for AArch64, introduced in Armv8.2-A. Unlike other SIMD architectures, SVE does not define the size of the vector registers, but constrains it to a range of possible values, from a minimum of 128 bits up to a maximum of 2048 in 128-bit wide units. The CPU designer can implement the extension by choosing the vector register size that best suits the workloads the CPU is targeting. The design of SVE guarantees that the same program can run on different implementations of the instruction set architecture without the need to recompile the code.

Many instructions use predicate registers to mask the lanes for operating on partial vectors. The SVE instruction set also provides gather loads and scatter stores, truncating stores, and signed/unsigned extended loads.

It contains the following sections:

2.1 What is the Scalable Vector Extension?

This topic is a brief introduction to Scalable Vector Extension (SVE).

Introduction

Scalable Vector Extension (SVE) is the next-generation SIMD extension of the *Armv8-A A64* instruction set. It is *not* an extension of *NEON*, but a new set of vector instructions developed to target HPC workloads. SVE enables vectorization of loops which would either be impossible or not beneficial to vectorize with NEON.

Unlike other SIMD architectures, SVE can be Vector Length Agnostic; it does does not fix the size of the vector registers leaving hardware implementors free to choose the size best suited to the intended workloads.

The SVE instruction set introduces the following new architectural features for High Performance Computing (HPC):

Scalable vector length

Vector code allows each implementation to automatically choose its vector length when it is a multiple of 128 bits and does not exceed the architectural maximum of 2048 bits. SVE provides 32 scalable vector registers, named Z0 – Z31.

Per-lane predication

SVE provides 16 predicate registers, named p0-p15, with each predicate register being 1/8th of the size of the vector register (1 bit per byte), and therefore scalable in size. Predicate registers are written to using condition-creating instructions, such as *compares*. This allows subsequent instructions to control which elements (or ‘lanes’) of a vector should be operated upon (the *active* elements).

Gather-load and scatter-store

Allows data to be efficiently transferred to or from a vector of non-contiguous memory addresses. This enables a significantly wider range of source code constructs to be vectorized. To permit efficient accesses to contiguous memory data, SVE provides a rich set of load and store instructions which progress sequentially forwards through an array, supporting a full range of packed 8, 16, 32 and 64-bit vector element organizations.

Vector partitioning

A vector partition is the dynamically determined portion of a vector defined by a predicate register. SVE permits the progression of a loop one partition at a time, until the whole vector has been processed or the loop has reached its natural conclusion.

Fault-tolerant speculative vectorization

Causes memory faults to be suppressed if they do not occur because of the first *active* element of the vector, and instead generates a predicate value indicating which of the requested lanes were successfully loaded prior to the first memory fault. This allows loops with conditional exits or unknown trip-counts to be safely vectorized, maintaining the same faulting behavior as if they had been executed sequentially. A common use for fault-tolerant speculative vectorization is in C strings.

Horizontal vector operations

SVE has a family of horizontal reduction instructions which include integer and floating-point summation, minimum, maximum, and bitwise logical reductions.

Serialized vector operations

SVE allows pointer chasing loops to be performed serially using a vector of addresses and an associated predicate, allowing the remainder of the loop to be parallelized.

Registers

The instruction set operates on a new set of vector and predicate registers:

- 32 `Z` registers, `z0`, `z1`, …, `z31`;
- 16 `P` registers, `p0`, `p1`, …, `p15`;
- 1 *First Faulting Register (FFR)* register.

The `Z` registers are data registers. The architecture specifies that their size in bits must be a *multiple of 128*, from a *minimum of 128 bits* to an implementation-defined maximum of up to 2048 bits. Data in these registers can be interpreted as 8-bit bytes, 16-bit halfwords, 32-bit words or 64-bit doublewords. For example, a 384-bit implementation of SVE can hold 48 bytes, 24 halfwords, 12 words, or 6 doublewords of data. It is also important to mention that the low 128 bits of each `Z` register overlap the corresponding *NEON* registers of the *Advanced SIMD* extension and therefore also the scalar floating-point registers:

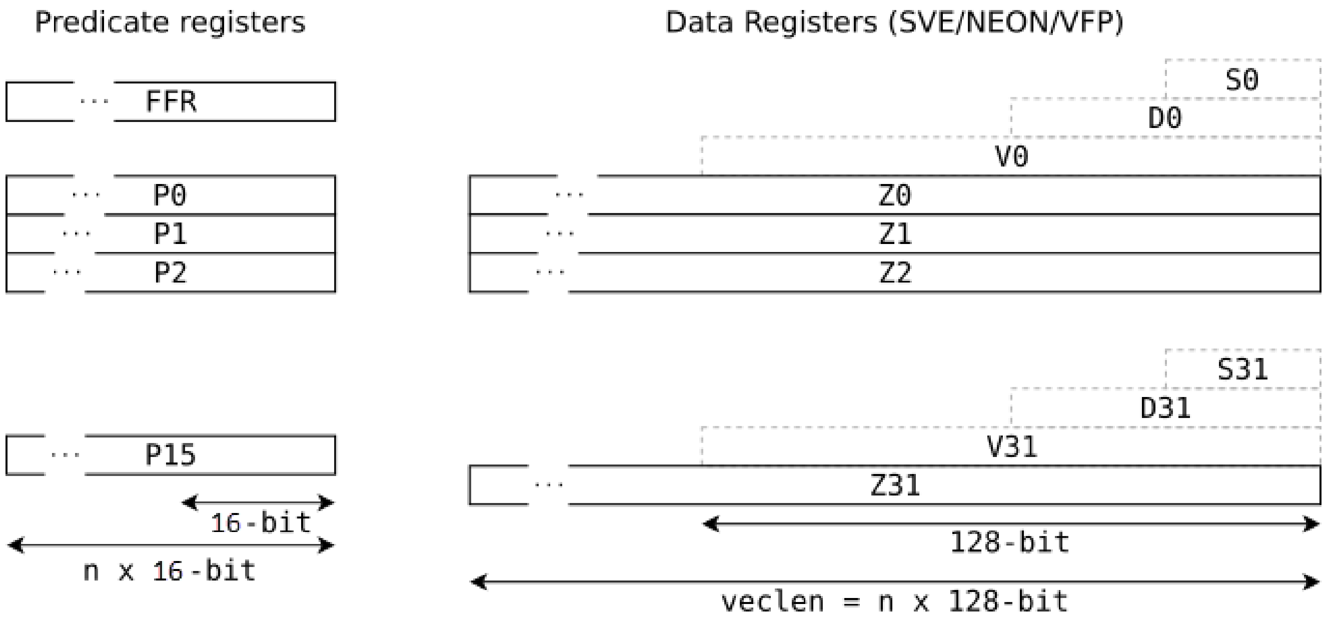

Figure 2-1 Register overlapping.

`P` registers are *predicate* registers, which are unique to SVE, and hold one bit for each byte available in a `Z` register. For example, an implementation providing 1024-bit `Z` registers provides 128-bit predicate registers.

The FFR register is a *special* predicate register that differs from regular predicate registers by way of being used implicitly by some dedicated instructions, called *first faulting loads*.

Individual predicate bits encode a boolean true or false, but a predicate *lane*, which contains between one and eight predicate bits is either *active* or *inactive*, depending on the value of its least significant bit.

Similarly, in this document the terms *active* or *inactive* lane will be used to qualify the lanes of data registers under the control of a predicate register.

Assembly language

The SVE assembly language is designed to closely mirror the AArch64 NEON mnemonics and operand syntax. However, SVE has significant differences which require extensions to the A64 assembly language, as follows:

New register files for vectors and predicates

Adds the register names z0-z31 and p0-p15.

Vector and predicate registers have unknown size

The element count is absent from a SVE vector or predicate shape suffix.

A predicate is a "bit mask"

SVE-capable assemblers report any inconsistencies between size suffixes and other operands as an error.

Zeroing or merging predication

Predicated instructions either zero the values of inactive lanes, 'zeroing form', or merge in the prior values, 'merging form'. These instructions have a suffix that indicates which form is being used.

Destructive encodings

Many instructions have destructive two-operand forms where the destination register also contains one of the source operands. To avoid ambiguity, the syntax uses a three-operand constructive notation, with the destructive operand being repeated in both the destination and source positions.

Gather-scatter addressing

The A64 load/store address syntax is extended to allow vector operands within the address specifier.

Predicate / vector condition codes

Adds a new set of aliases for condition codes for use in SVE assembler source and disassembly.

SVE instruction set

SVE introduces a variety of instructions that operate on the data and predicate registers. There are two main classes of instructions; *predicated* and *unpredicated*. Instructions that use a predicate register to control the lanes they operate on, versus those that do not. In a predicated instruction, only the active lanes of vector operands are processed and can generate side effects - such as memory accesses and faults, or numeric exceptions.

Across these two main classes, there are *data processing* instructions, that operate on Z registers (for example, addition), *predicate generation* instructions, such as numeric comparisons that operate on data registers and produce predicate registers, or *predicate manipulation* instructions, that mostly cover predicate generation or logical operations on predicates.

Note

Only predicate registers p0 through p7 are usable as predicates in data-processing instructions.

Most data manipulation operations cover both floating point (FP) and integer domains, with some notable FP functionality brought by the *ordered horizontal reductions*, which provide cross-lane operations that preserve the strict C/C++ rules on non-associativity of floating-point operations.

A significant proportion of the new instruction set is dedicated to *vector load/store* instructions, which can perform *signed* or *unsigned extension* or *truncation* of the data, and come with a wide range of new addressing modes that improve the efficiency of SVE code.

SVE instructions can be separated by function into the following groups. For a more detailed description of the instructions, see the *ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A* document.

Load, store, and prefetch instructions

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate registers. SVE also includes vector prefetch instructions that provide read and write hints to the memory system. Instructions include:

- Predicated single vector contiguous element accesses
- Predicated non-contiguous element accesses
- Predicated multiple vector contiguous structure load/store
- Predicated replicating element loads
- Unpredicated vector register load/store
- Unpredicated predicate register load/store

——— **Note** ———

Unpredicated vector register load/store do not have endianness conversion, and should not be used for your code.

Vector move operations

Vector move instructions copy data from scalar registers, immediate values, and other vectors to selected vector elements. Instructions include:

- Element move and broadcast

Integer operations

The integer instructions operate on signed or unsigned integer data within a vector. Instructions include:

- Integer arithmetic
- Integer dot product
- Integer comparisons

Vector address calculation

The vector address calculation instructions compute vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current vector length or predicate register length, in bytes, to a general-purpose register.

Bitwise operations

The bitwise instructions perform bitwise operations on vectors. Instructions include:

- Bitwise shift, reverse, and count

Floating-point operations

The floating-point instructions operate on floating-point data within a vector. Instructions include:

- Floating-point arithmetic
- Floating-point multiply accumulate
- Floating-point complex arithmetic
- Floating-point rounding and conversion
- Floating-point comparisons
- Floating-point transcendental acceleration
- Floating-point indexed multiplies

Predicate operations

The predicate instructions relate to operations that manipulate the predicate registers. Instructions include:

- Predicate initialization
- Predicate move operations
- Predicate logical operations
- FFR predicate handling
- Predicate counts
- Loop control
- Serialized operations

Move operations

These instructions move data between different vector elements, or between vector elements and scalar registers. Instructions include:

- Element permute and shuffle
- Unpacking instructions
- Predicate permute
- Index vector generation
- Move prefix

Reduction operations

Horizontal reduction instructions perform arithmetic horizontally across *active* elements of a single source vector and deliver a scalar result. Instructions include:

- Horizontal reductions

Arm C Language Extensions (ACLE) for Arm SVE

The aim of the Arm C language extensions (ACLE) is to make features of the Arm architecture directly available in C and C++ programs. The core ACLE is defined in a *dedicated document*, while the *ACLE for Arm SVE* document defines the part that is specific to the Arm Scalable Vector Extension (SVE).

General information on Arm C Language Extensions is available on the *ACLE Developer web page*.

2.2 Why is the Scalable Vector Extension (SVE) useful for HPC?

Using the new architectural features, SVE moves beyond NEON's traditional strengths in data-plane processing. SVE not only offers wider vectors, but also provides the following benefits:

- Vectorization techniques can be applied across a wider range of program loops containing complex control flows and data structures.
- Lowering the cost of development for SIMD code that might traditionally have required hand coding.
- Greater fine-grain data parallelism in real-world programs.

Why this is useful for HPC applications?

Rather than specifying a vector length, SVE allows CPU designers to choose the most appropriate vector length for their application and market, from 128 bits up to 2048 bits per vector register. SVE also supports an auto vector-length agnostic (VLA) programming model that can adapt to the available vector length. Adopting this method means you can compile or hand-code your program for SVE hardware once, while avoiding the need to recompile or rewrite it when longer vectors appear in the future. This reduces deployment costs over the lifetime of the architecture; a program just works 'everywhere' and executes wider and faster. Importantly, this makes programming easier when developing and porting code; ensuring better scalability and compatibility into the future.

Scientific workloads are carefully written to exploit as much data-level parallelism as possible, making careful use of *OpenMP* pragmas and other source code annotations. Therefore, it is relatively straightforward for a compiler to vectorize such code and make good use of a wider vector unit, if it is available. To the benefit of SVE, HPC clusters are also built with the wide, high-bandwidth memory systems that are necessary to feed a longer vector unit. Furthermore, SVE's instructions are designed to support full floating-point features that are compliant to IEEE754 standard, including half-precision floating-point.

Related information

New architectural features

2.3 Additional resources

For a more detailed description of SVE, why it is useful for HPC, or to see some example code, see the following additional resources:

Want to evaluate SVE?

Download *Arm Compiler for HPC* and *Arm Instruction Emulator* to compile and run SVE code on non-SVE platforms.

For a more detailed description of SVE, why it is useful for HPC, or to see some example code, see the following additional resources:

- *SVE hackathon SC18*

  An SVE emulator tutorial and the example code from the SC18 hackathon.
- *White paper: A sneak peek into SVE and VLA programming*

  An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.
- *Arm Scalable Vector Extensions and application to machine learning*

  In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of machine learning system. These examples are written with the Vector Length Agnostic (VLA) approach introduced by the Scalable Vector Extension (SVE).
- *Arm C Language Eextensions for SVE*

  The SVE ACLE defines a set of C and C++ types and accessors for SVE vectors and predicates.
- *DWARF for the ARM® 64-bit Architecture (AArch64) with SVE support*

  This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.
- *Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support*

  This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.
- *Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*

  This supplement describes the Scalable Vector Extension to the Armv8-A architecture profile.
- *Arm Instruction Emulator*

  Arm Instruction Emulator (ArmIE) runs on AArch64 platforms and emulates SVE instructions.
- *Arm HPC tools*

  Learn more about the compilers, math libraries, debugging, and profiling tools Arm offers.

Chapter 3
SVE Vector Length Agnostic programming

This chapter introduces the concept of Vector Length Agnostic (VLA) programming and provides SVE programming tips supported by examples in assembly that use the Arm C Language Extensions (ACLE) for SVE.

It contains the following sections:

3.1 Vector Length Agnostic (VLA) programming

Unlike traditional SIMD architectures which define a fixed size for their vector registers, SVE only specifies a maximum size. This enables different Arm architectural licensees to develop their own implementation, targeting specific workloads and technologies which could benefit from a particular vector length.

A key goal of SVE is to allow the same program image to be run on any implementation of the architecture, which might implement different vector lengths.

These features require a new programming style, called *Vector Length Agnostic (VLA) programming*.

If you want to vectorize the loop inside the function `example01` of *example01*. With a traditional SIMD architecture, the user (or the compiler) knows how many elements the vector loop can process in one iteration. For example, the vectorized version of `example01` can be rewritten as `example_01_neon` in *example01_neon*, using *Arm NEON* (*C intrinsics*). Here, the loop operates on four elements. In other words, as many 32-bit `ints` as a NEON vector register can hold. Moreover, as in the case of other traditional unpredicated SIMD architectures, the programmer (or the compiler) needs to add an additional loop, called *loop tail*, that is responsible for processing those iterations at the end of the loop that do not fit in a full vector length.

Simple C loop processing integers (*example01*):

```
void example01(int *restrict a, const int *b, const int *c, long N)
{
  long i;
  for (i = 0; i < N; ++i)
    a[i] = b[i] + c[i];
}
```

Vectorized version of *example01*, using NEON C intrinsics (*example01_neon*):

```
void example01_neon(int *restrict a, const int *b,
                    const int *c, long N)
{
  long i;
  // vector loop
  for (i = 0; i < N - 3; i += 4) {
    int32x4_t vb = vld1q_s32(b + i);
    int32x4_t vc = vld1q_s32(c + i);
    int32x4_t va = vaddq_s32(vb, vc);
    vst1q_s32(a + i, va);
  }
  // loop tail
  for (; i < N; ++i)
    a[i] = b[i] + c[i];
}
```

With SVE, the *fixed-width* approach is not appropriate. In *example01_sve* an assembly version of `example01` is shown, with SVE instructions. The assembly code is equivalent to the pseudo-C code presented in *example01_sve_pseudo_c*, where the `loop_body` section is repeated if the condition `cond` is true. The condition is tested on the predicate register `p0` that is created using the `whilelt` instruction. The following example shows in detail how it works.

VLA SVE code for *example01* - (*example01_sve*):

```
    # x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
    mov     x4, 0 # set 'i=0'
    b       cond  # branch to 'cond'
loop_body:
    ld1w    z0.s, p0/z, [x1, x4, lsl 2] # load vector z0 from address 'b + i'
    ld1w    z1.s, p0/z, [x2, x4, lsl 2] # same, but from 'c + i' into vector z1
    add     z0.s, p0/m, z0.s, z1.s      # add the vectors
    st1w    z0.s, p0, [x0, x4, lsl 2]   # store vector z0 at 'a + i'
    incw    x4                          # increment 'i' by number of words in a vector
cond:
    whilelt p0.s, x4, x3  # build the loop predicate p0, as p0.s[idx] = (x4+idx) < x3
                          # it also sets the condition flags
    b.first    loop_body  # branch to 'loop_body' if the first bit in the predicate
                          # register 'p0' is set
    ret
```

In the assembly code, `x4` corresponds to the value of the loop induction variable `i` and `x3` is the loop bound variable `N`. The assembly line `whilelt p0.s, x4, x3` is fills the predicate register `p0` by setting each lane as `p0.s[idx] := (x3 + idx) < x4` (`x3` and `x4` hold `i` and `N` respectively), for each of the indexes `idx` corresponding to 32-bit lanes of a vector register. For example, *predicate* shows the content of `p0` that `whilelt` generates in case of a 256-bit SVE implementation for `N=7`.

Example of predicate register with 32-bit lanes view (*predicate*):

```
P0 = [0000 0001 0001 0001 0001 0001 0001 0001]
         7    6    5    4    3    2    1    0  32-bit lanes 'idx'
```

When building the predicate `p0`, the `whilelt` also sets the condition flags. The branching instruction `b.first` following `whilelt` reads those flags and decides whether or not to branch to the `loop_body` label. In this specific case, `b.first` checks whether the first (LSB) lane of `p0.s` is set to true, that is, whether there are any further elements to process in the next iteration of the loop. Notice that the concept of lanes refers to the element size specifier used in the condition setting instruction, as the example in *predicate* shows. SVE provides many conditions that can be used to check the condition flags. For example, `b.none` checks if all the predicate lanes have been set to false, `b.last` checks if the last lane is set to true, `b.any` if any of the lanes of the predicate are set to true. Notice that concepts like *first* and *last* in the vector requires the introduction of an ordering. The list of instructions that can be used to test the condition flags set by SVE instructions is shown below.

SVE branching instruction testing condition flags:

Branch instruction	**SVE interpretation**
`b.none`	No active elements are true.
`b.any`	An active element is true.
`b.nlast`	The last active element is not true.
`b.last`	The last active element is true.
`b.first`	The first active element is true.
`b.nfirst`	The first active element is not true.
`b.pmore`	An active element is true but not the last element.
`b.plast`	The last active element is true or none are true.
`b.tcont`	Scalarized CTERM loop termination not detected.
`b.tstop`	Scalarized CTERM loop termination detected.

The assembly example in *example01_sve* is equivalent to *example01_sve_pseudo_c*.

C pseudo code for the blocks in *example01_sve* - (*example01_sve_pseudo_c*):

```
while( /* cond */ ) {
  /* loop */
}
```

The loop body is also vector length agnostic. In each iteration, the operations performed are:

- load two vectors of data from `b` and `c` respectively, with `ld1w`. Here, the loads use *register plus register addressing mode*, where the index register `x4` is left-shifted by two bits to scale the index by 4, corresponding to the size of the scalar data;
- add the values into another register with the `add` instruction;
- store the value computed into `a` with `st1w` (same register plus register addressing as `ld1w`).
- increment the index `x4` by a number of 32-bit words in the machine-implemented vector length with `incw`.

All the instructions in the `loop_body` are predicated with `p0`, meaning that inactive 32-bit lanes are not accessed by the instruction, so that the scalar loop tail is not needed.

The `incw` instruction is an important VLA feature used in the code. This loop executes correctly on any implementation of SVE, because `incw` takes care of increasing the loop iterator according to the current SVE vector length.

For the purpose of comparison, an assembly version of the NEON code in *example01_neon* is shown in *example01_neonassembly*, and shows both the vector body and the loop tail blocks.

NEON assembly code generated from the C intrinsics in *example01_neon* - (*example01_neonassembly*).

```
    # x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x8 is the loop induction variable 'i'
    mov     x8, xzr
    subs    x9, x3, 3                 # x9 = N -3
    b.ls    .loop_tail_preheader      # jump to loop tail if N <= 3
.vector_body:
    ldr     q0, [x1, x8, lsl 4]       # load 4 elements from 'b+i'
    ldr     q1, [x2, x8, lsl 4]       # load 4 elements from 'c+i'
    add     v0.4s, v1.4s, v0.4s       # add the vector
    str     q0, [x0, x8, lsl 4],      # store 4 elements in 'a+i'
    add     x8, x8, 4                 # increment 'i' by 4
    cmp     x8, x9                    # compare i with N - 3
    b.lo    .vector_body              # keep looping if i < N-3
.loop_tail_preheader:
    cmp     x8, x3                    # compare the loop counter with N
    b.hs    .function_exit            # if greater or equal N, terminate
.loop_tail:
    ldr     w12, [x1, x8, lsl 2]
    ldr     w13, [x2, x8, lsl 2]
    add     w12, w13, w12
    str     w12, [x0, x8, lsl 2]
    cmp     x8, x3
    b.lo    .loop_tail                # keep looping until no elements remain
.function_exit:
    ret
```

More on predication

Predication can also be used to vectorize loops with control flow in the loop body. The `if` statement of *example02* is executed in the vector `loop-body` of the code in *example02_sve* by setting the predicate `p1` with the `cmpgt` instruction, which tests for *compare greater than*. This operation produces a predicate that selects which lanes have to be operated by the if-guarded instruction. The loads and the stores of the data in `a`, `b` and `c` arrays are performed by instructions that use the predicate `p1`, so that the only elements in memory that are modified correspond to those modified by the original C code.

A loop with conditional execution (*example02*):

```
void example02(int *restrict a, const int *b, const int *c, long N,
                      const int *d)
{
  long i;
  for (i = 0; i < N; ++i)
    if (d[i] > 0)
      a[i] = b[i] + c[i];
}
```

SVE vector version of *example02* - (example02_sve*).

Note

The comments in the assembly code relate only to the predication specific behavior that is shown.

```
    # x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'd', x5 is 'i'
    mov     x5, 0 # set 'i = 0'
    b       cond
loop_body:
    ld1w    z4.s, p0/z, [x4, x5, lsl 2] # load a vector from 'd + i'
    cmpgt   p1.s, p0/z, z4.s, 0         # compare greater than zero
                                        # p1.s[idx] = z4.s[idx] > 0
    # from now on all the instructions depending on the 'if' statement are
    # predicated with 'p1'
    ld1w    z0.s, p1/z, [x1, x5, lsl 2]
    ld1w    z1.s, p1/z, [x2, x5, lsl 2]
    add     z0.s, p1/m, z0.s, z1.s
    st1w    z0.s, p1, [x0, x5, lsl 2]
    incw    x5
cond:
    whilelt p0.s, x5, x3
    b.ne    loop_body
    ret
```

Merging and zeroing predication

Some of the data processing instructions have two different kinds of predication, *merging* and *zeroing*. Merging is indicated by the `/m` qualifier attached to the instruction's governing predicate, as in `add z0.s, p1/m, z0.s, z1.s`; zeroing is indicated by the `/z` qualifier, as in `cmpgt p1.s, p0/z, z4.s, 0`.

Merging and zeroing predication differ in the way the instruction operates on inactive lanes. Zeroing sets the inactive lanes to zero, while merging leaves the inactive lanes unchanged.

The examples in *example04_c* and *example04_sve* show how merging predication can be used to perform a conditional reduction.

A reduction (*example04_c*):

```
int example02(int *a, int *b, long N)
{
  long i;
  int s = 0;
  for (i = 0; i < N; ++i)
    if (b[i])
      s += a[i];
  return s;
}
```

SVE vector version of *example04_c* - (*example04_sve*):

```
     mov     x5, 0   # set 'i = 0'
     mov     z0.s, 0 # set the accumulator 's' to zero
     b       cond
loop_body:
     ld1w    z4.s, p0/z, [x1, x5, lsl 2] # load a  vector
                                         # at 'b + i'
     cmpne   p1.s, p0/z, z4.s, 0         # compare non zero
                                         # into predicate 'p1'
     # from now on all the instructions depending on the 'if' statement are
     # predicated with 'p1'
     ld1w    z1.s, p1/z, [x0, x5, lsl 2]
     add     z0.s, p1/m, z0.s, z1.s      # the inactive lanes
                                         # retain the partial sums
                                         #  of the previous iterations
     incw    x5
cond:
     whilelt p0.s, x5, x3
     b.first loop_body
     ptrue p0.s
     saddv d0, p0, z0.s # signed add words across the lanes of z0, and place the
                        # scalar result in d0
     mov w0, v0.s[0]
     ret
```

Gather loads

Another important feature introduced with SVE is the *gather load* / *scatter store* set of instructions, which allow it to operate on non-contiguous data in memory, like the code shown in *example03*.

Loading data from an array of addresses (*example03*):

```
void example03(int *restrict a, const int *b, const int *c,
               long N, const int *d)
{
  long i;
  for (i = 0; i < N; ++i)
      a[i] = b[d[i]] + c[i];
}
```

The vector code in *example03_sve* uses a special version of the `ld1w` instruction to load the data at `b[d[i]]`, `ld1w z0.s, p0/z, [x1, z1.s, sxtw 2]`. The values stored in `z1.s` are interpreted as 32-bit scaled indices, and sign extended (`sxtw`) to 64-bit before being left-shifted by 2 and added to the base address `x4`. This addressing mode is called *scalar plus vector* addressing mode. This code shows only one example of it. To account for many other situations that occur in real world code, other addressing modes support a 32-bit unsigned index or a 64-bit index, with and without scaling.

SVE vectorized version of *example03* - (*example03_sve*):

```
     mov     x5, 0
     b       cond
loop:
     ld1w    z1.s, p0/z, [x4, x5, lsl 2]
     ld1w    z0.s, p0/z, [x1, z1.s, sxtw 2] # load a vector
                                            # from  'x1 + sxtw(z1.s) << 2'
     ld1w    z1.s, p0/z, [x2, x5, lsl 2]
     add     z0.s, p0/m, z0.s, z1.s
     st1w    z0.s, p0, [x0, x5, lsl 2]
     incw    x5
cond:
     whilelt p0.s, x5, x3
     b.first    loop
     ret
```

Related information

Arm C Language Extensions for SVE

3.2 For and While loop vectorization

The following code is an example of a simple vectorization of `For` and `While` loops. The code shows the difference in the vectorization approach between the NEON instruction set (with fixed vector length), and the vector length agnostic SVE instruction set.

A simple `for` example that computes the addition of two arrays of integers:

```
void example_for( int *restrict out, int *restrict a, int *restrict 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 arrays of integers:

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

Note

Both code examples presented above are doing the same addition of two arrays of integers.

For/While loop NEON vectorization

```
1       AND w6, w3, 0xfffffffc
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 the above example, the working element size is 32-bits. Four 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 handle array lengths that are not multiples of 4, you must implement the scalar loop to calculate the remaining results if there are any (the loop at lines 16-22).

For/While loop SVE vectorization

The SVE Vector Length Agnostic (VLA) vectorization approach involves 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 is 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
```

Like the Advanced SIMD vectorization, the processing lane width is 32-bits. On line 4, the governing predicate `p1` initializes for 32-bit elements with the instruction `WHILELT`. The `WHILELT` instruction 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:

1. Accessed from the memory and loaded into the vector registers (lines 8-9)
2. Processed (line 10)
3. 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 as the load and store pointer offset. Each vectorized loop iteration is incremented in the vector length agnostic approach. With the instruction `INCW`, this value is incremented by the number of 32-bit elements in the implemented vector length (line 12).

Note

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 the `WHILELT` instruction (line 13). If there is at least one more input array element to process, the next loop iteration occurs. Otherwise, the loop exits. The conditional branch `B.FIRST` (line 14), manages whether there is another input array element to process.

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

```
void example_for( int *out, int *a, int *b, int N) {
    uint64_t i = 0;
    uint64_t vl = svcntw();
    svbool_t pred;
    svint32_t sva, svb, svres;

    pred = svwhilelt_b32( i, (uint64_t)N);

    while(svptest_first(svptrue_b32(), pred)) {
        sva = svld1( pred, &a[i]);
        svb = svld1( pred, &b[i]);
        svres = svadd_m( pred, sva, svb);
        svst1( pred, &out[i], svres);
        i += vl;
        pred = svwhilelt_b32( i, (uint64_t)N);
    }
}
```

For/While loop SVE vectorization with prefetching

The execution speed of the optimized loop from the previous section can be improved by using the prefetch instructions. Prefetch instruction signals to the memory system to preload 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` (lines 18-19). The memory system takes actions to preload the specified memory addresses to L1 cache `PLDL1STRM`. The prefetch instruction also signals to the memory system that these memory addresses will be used only once, and that there is no need to keep them in the local cache `PLDL1STRM`.

In this example, all true predicate `p0` is used with prefetch instructions. The result is that an additional 3 vector lengths of data, just after both of the input arrays, will be prefetched, but not loaded. For certain applications where these redundant prefetches could cause a problem, Arm recommends setting predicate `p0` independently for each vector length prefetch, with the appropriate `WHILELT` instruction in the inner loop.

3.3 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.

The *Do-while* loop that computes the 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 of 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`.

To begin, for each vector lane, the starting input element is set in a Vector Length Agnostic (VLA) approach with instruction `INDEX` (line 5).

`CMPGT` (line 6), partitions the vector and sets the loop process-governing predicate `p0`. The lanes holding positive elements are set to active, while the remaining 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` also act as the loop counter (in vector form). The next group of input elements is computed in a Vector Length Agnostic (VLA) approach, with instruction `DECW` (line 11). To obtain the input element of that lane for the next loop iteration, the current input element in each vector lane is decremented by the number of 32-bit elements in the implemented vector length. Then, with instruction `CMPGT` (line 12), the new vector partitions are based on the newly-calculated input elements. If at least one positive input element exists, the next loop iteration occurs. Otherwise, the loop exits. The conditional branch `B.FIRST` (line 13), checks if at least one positive input element exists.

To finish, the reduction instruction `UADDV` (line 15), calculates the final sum outside of the loop. The partial sums from all lanes of vector register `z5` are summed, and controlled by the predicate `p1`, which 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* as:

```
1   void example_sum_squares( int N, int * sum) {
2       svbool_t pred_N;
3       svint32_t svN_tmp;
```

```
    svbool_t p_all = svptrue_b32();
    svint32_t acc = svdup_s32(0);

    if (N > 0) {
        svN_tmp = svindex_s32( N, -1);
        pred_N = svcmpgt( p_all, svN_tmp, 0);

        do {
            acc = svmla_m( pred_N, acc, svN_tmp, svN_tmp);
            svN_tmp = svsub_x( p_all, svN_tmp, svcntw());
            pred_N = svcmpgt( p_all, svN_tmp, 0);
        } while ( svptest_first( p_all, pred_N));
    }

    *sum = (int) svaddv( p_all, acc);
}
```

3.4 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 (VLA) approach. To achieve a maximally filled vector length, often, in a vector register, you must arrange data in a specific way. Sometimes, to arrange the data in a specific way, you are also required to unroll the loop.

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 reverse order.

The SVE vectorized implementation:

```
    MOV x5, #0
    SXTW x0, w0
    WHILELT p4.s, x5, x0
    B.NONE .L_return

    SXTW x1, w1
    ADD x1, x3, x1, LSL #1
    PTRUE p5.s

.L_OuterLoop:
    MOV x6, #0
    MOV x7, x3
    LD1SH z10.s, p4/Z, [x2, x6, LSL #1]
    LD1RSH z1.s, p5/Z, [x7]
    ADD x6, x6, #1
    ADD x7, x7, #2
    MUL z10.s, p4/M, z10.s, z1.s
    CMP x7, x1
    B.EQ .L_InnerLoopEnd

.L_InnerLoop:
    LD1SH z2.s, p4/Z, [x2, x6, LSL #1]
    LD1RSH z1.s, p5/Z, [x7]
    ADD x6, x6, #1
    ADD x7, x7, #2
    MLA z10.s, p4/M, z2.s, z1.s
    CMP x7, x1
    B.MI .L_InnerLoop

.L_InnerLoopEnd:
    ASR z10.s, p4/M, z10.s, #16
    ST1H z10.s, p4, [x4]
    INCH x4
    INCH x2
    INCW x5
    WHILELT p4.s, x5, x0
    B.FIRST .L_OuterLoop

```

```
.L_return:
    RET
```

Chapter 4
Generic Vector and Matrix Operations Examples

This chapter discusses generic vector and matrix operations examples.

It contains the following sections:

4.1 Vector Maximum Element

This section looks at a code example that finds the maximum element of a vector.

This section contains the following subsection:
- *4.1.1 Vector Maximum with Real Fixed-Point 16-bit Elements* on page 4-35.

4.1.1 Vector Maximum with Real Fixed-Point 16-bit Elements

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

The following C code example shows how to find the maximum element and its first location in a vector with real fixed-point 16-bit elements:

```
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 code example 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`. The 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 occurence 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`. The indexes corresponding to other non-maximum vector elements are overwritten by the value `0xFFFF`. Finally, the first occurence 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.

Related concepts

4.1.1 Vector Maximum with Real Fixed-Point 16-bit Elements on page 4-35

4.2 Vector Dot-Product

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

This section contains the following subsections:

- *4.2.1 Vector Dot-Product Calculation* on page 4-37.
- *4.2.2 Vectors Dot-Product with Complex SP Floating-Point Elements and Result* on page 4-37.

4.2.1 Vector Dot-Product Calculation

Vector dot-product is computed as:

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

The following implementations are based on complex data that are organized in a vector register with the real part in even vector elements, and the imaginary parts in the corresponding odd vector elements.

4.2.2 Vectors Dot-Product with Complex SP Floating-Point Elements and Result

The following example is the optimized SVE implementation of dot-product computation of two vector elements 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`). The instructions `B.NFRST` (line 13) `B.FIRST` (line 29), check whether there are more than one vector length input elements remaining, and only in the case where the vectorized loop starting (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
```

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 is first `FCMLA` instruction with rotation parameter `#0`. For two complex-valued inputs:

$$\alpha = \alpha_r + i * \alpha_i, \beta = \beta_r + i * bi$$

the first `FCMLA` instruction produces the following result:

$$c_r + = \alpha_r * \beta_r, c_i + = \alpha_r * \beta_i$$

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

$$c_r + = -\alpha_i * \beta_i, c_i + = \alpha_i * \beta_r$$

Therefore,

$$c + = (\alpha_r * \beta_r - \alpha_i * bi) + i * (\alpha_r * \beta_i + \alpha_i * \beta_r)$$

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 resulting real and imaginary parts interleaved.

To compute the final two vector elements dot-product result, you must perform the reduction addition. More precisely, to compute the real part of the final result, you must perform the reduction addition of partial real part results only. Similarly, to compute the imaginary part of the final result, you must perform the reduction addition of partial imaginary part results only.

The real and imaginary parts of the partial sums are de-interleaved, using instructions `UZP1` and `UZP2` (lines 43-44). The partial real part sums are moved to register `z10`, and the partial imaginary part sums are moved to register `z11`. The reduction additions are computed with the instruction `FADDV`, for the real part (line 45), and for the imaginary part (line 46). A real-imaginary pair; one complex SP floating-point result is stored to the memory by instruction `STP` (line 47).

Related concepts

4.2.1 Vector Dot-Product Calculation on page 4-37

4.2.2 Vectors Dot-Product with Complex SP Floating-Point Elements and Result on page 4-37

4.3 FIR Filter

This section provides code examples of Finite Impulse Response (FIR) filtering computation.

This section contains the following subsections:

- *4.3.1 FIR Filtering Theory and C Implementation* on page 4-39.
- *4.3.2 FIR filtering with real SP floating-point elements* on page 4-39.

4.3.1 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:

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

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

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

```
#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
}
```

4.3.2 FIR filtering with real SP floating-point elements

The following example 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]
```

```
    ADD x6, x6, #1
    ADD x8, x8, #4
    FMUL z10.s, z2.s, z1.s

.L_FIR_inner_loop:
    LD1W z2.s, p4/z, [xPtr, x6, LSL #2]
        // contiguous load of input data
    LD1RW z1.s, p5/z, [x8]
        // load with broadcast of one FIR filter coefficient
    ADD x6, x6, #1
    ADD x8, x8, #4
    FMLA z10.s, p5/m, z2.s, z1.s
    CMP x8, taps
    B.MI .L_FIR_inner_loop

    ST1W z10.s, p4, [yPtr]
    INCB yPtr
    ADDVL xPtr, xPtr, #1
    WHILELT p4.b, yPtr, size
    B.FIRST .L_FIR_outer_loop

.L_return:
    RET
```

Related concepts

4.3.1 FIR Filtering Theory and C Implementation on page 4-39

4.3.2 FIR filtering with real SP floating-point elements on page 4-39

4.4 Matrix Multiplication

This section provides code examples of matrix multiplication.

This section contains the following subsections:
- *4.4.1 Matrix Multiplication with Real DP Floating-Point Elements* on page 4-41.
- *4.4.2 Matrix Multiplication with Real HP Floating-Point Elements* on page 4-42.
- *4.4.3 Matrix Multiplication with Real Fixed-Point 8-Bit Input Elements and Real Fixed-Point 32-Bit Output Elements* on page 4-46.

4.4.1 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 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

This code is based on the input and output matrices organized in memory in a row-major order.

Code example

The following SVE vectorization prerequisites apply:

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

The real DP floating-point matrix multiplication function is written in C with the *ARM C Language Extensions for SVE*. The following example is a vector length agnostic implementation, for vector lengths that are a powers of 2, and up to 1024-bits supported.

```
void matmul_f64( uint64_t M, uint64_t K, uint64_t N,
                float64_t * inLeft, float64_t * inRight, float64_t * out) {
     uint64_t x, y, z;
     svbool_t p64_all = svptrue_b64();
     uint64_t vl = svcntd();
     uint64_t offsetIN_1, offsetIN_2, offsetIN_3;
     uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3;

     float64_t *ptrIN_left;
     float64_t *ptrIN_right;
     float64_t *ptrOUT;

     svfloat64_t acc0, acc1, acc2, acc3;
     svfloat64_t inR_0, inR_1;
     svfloat64_t inL_0, inL_1, inL_2, inL_3;

     offsetIN_1 = K;
     offsetIN_2 = 2*K;
     offsetIN_3 = 3*K;

     offsetOUT_1 = N;
     offsetOUT_2 = 2*N;
     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 innermost 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 loaded (two elements per one vector load) and replicated by instruction `LD1RQD` (lines 41-44). Right-input matrix elements are loaded by the contiguous vector length load `LD1D` (lines 38-39). The results of the innermost loop completion subblock, `out[4, vl]`, are stored to the memory by the contiguous vector length store `ST1D` (lines 62-65).

4.4.2 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;
        }
    }
}
```

This code is based on all matrices organized in memory in a row-major order.

Code example

The following SVE vectorization prerequisites apply:

- 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 for SVE*. The implementations are vector length agnostic, for the vector lengths that are powers of 2 and up to 1024-bits supported.

The following is the SVE implementation of left-matrix rearrangement:

```
void rearrangeLeft_fp16( uint64_t M, uint64_t K,
                         float16_t * inLeft, float16_t * inLeft_MOD) {
    uint64_t x, y, nb_st_elems, init_nb_elems;
    svbool_t p_ld;
    svfloat16_t r0, r1, r2, r3, r4, r5, r6, r7;
    uint64_t offsetIN_1, offsetIN_2, offsetIN_3, offsetIN_4;
    uint64_t offsetIN_5, offsetIN_6, offsetIN_7;

    float16_t *ptrIN;
    float16_t *ptrOUT;

     uint64_t vl = svcnth();
     svbool_t p16_all = svptrue_b16();

     offsetIN_1 = K;
     offsetIN_2 = 2*K;
     offsetIN_3 = 3*K;
     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 the 8 matrix rows is implemented using the `ZIP1` and `ZIP2` instructions, in 3 steps (lines 42-67). Because the 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 last iteration of the inner loop might have inactive lanes. Therefore, the rearranged matrix elements stores to the memory are managed by the switch statement (lines 69-81).

The following is an SVE implementation of a matrix multiplication results computation:

```
void matmul_dotp_fp16( uint64_t M, uint64_t K, uint64_t N,
                       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 = svld1rq(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 innermost 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 innermost loop completion subblock `out[8, vl]`, results are stored to the memory by contiguous vector length store `ST1H` (lines 56-63).

Since the matrix dimensions are a 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.

4.4.3 Matrix Multiplication with Real Fixed-Point 8-Bit Input Elements and Real Fixed-Point 32-Bit Output Elements

The fixed-point matrix multiplication, with real 8-bit input elements and real 32-bit output elements:

$$out[M,N] = inLeft[M,K] * inRight[K,N]$$

is implemented in two steps. In the first step, both the left and right matrices are rearranged in a specific method, 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:

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

These rearrangements are 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. Next, 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.

The calculations are 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;
        }
    }
}
```

This code is based on all matrices organized in memory in a row-major order.

Code example

The following SVE vectorization prerequisites apply:

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

The fixed-point matrix multiplication with real 8-bit input elements and real 32-bit output elements functions are written in C with the *ARM C Language Extensions for SVE*. For the vector lengths that are powers of 2 and up to 1024-bits supported, the implementations are vector length agnostic.

The following code is an SVE implementation of the left-matrix rearrangement:

```
void rearrangeLeft_fixp( uint64_t M, uint64_t K,
                         uint8_t * inLeft, uint8_t * inLeft_MOD) {
    uint64_t x, y, nb_st_elems, init_nb_elems;
    svbool_t p_ld;
    uint64_t offsetIN_1, offsetIN_2, offsetIN_3, offsetIN_4;
    uint64_t offsetIN_5, offsetIN_6, offsetIN_7;

    svuint8_t r0, r1, r2, r3, r4, r5, r6, r7;
    svuint32_t r00, r11, r22, r33, r44, r55, r66, r77;
     svuint32_t t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19;

     uint8_t *ptrIN;
     uint8_t *ptrOUT;

     uint64_t vl = svcntb();
     svbool_t p8_all = svptrue_b8();

     offsetIN_1 = K;
     offsetIN_2 = 2*K;
     offsetIN_3 = 3*K;
     offsetIN_4 = 4*K;
     offsetIN_5 = 5*K;
     offsetIN_6 = 6*K;
     offsetIN_7 = 7*K;

     init_nb_elems = 8*K/vl;

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

        nb_st_elems = init_nb_elems;

        for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, K)); y+=vl) {
            r0 = svld1(p_ld, ptrIN);
            r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
            r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
            r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
            r4 = svld1(p_ld, &ptrIN[offsetIN_4]);
            r5 = svld1(p_ld, &ptrIN[offsetIN_5]);
            r6 = svld1(p_ld, &ptrIN[offsetIN_6]);
            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                      svst1_vnum(p8_all, ptrOUT, 7, svreinterpret_u8(r77));
74                      svst1_vnum(p8_all, ptrOUT, 6, svreinterpret_u8(r66));
75                      svst1_vnum(p8_all, ptrOUT, 5, svreinterpret_u8(r55));
76                      svst1_vnum(p8_all, ptrOUT, 4, svreinterpret_u8(r44));
77                  case 4 :
78                      svst1_vnum(p8_all, ptrOUT, 3, svreinterpret_u8(r33));
79                      svst1_vnum(p8_all, ptrOUT, 2, svreinterpret_u8(r22));
80                  case 2 :
81                      svst1_vnum(p8_all, ptrOUT, 1, svreinterpret_u8(r11));
82                      svst1(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 the left-matrix involves grouping matrix elements into sets of four consecutive elements, and transposing multiple 8-rows blocks of sets. This rearrangement is implemented with `ZIP1` and `ZIP2` instructions in 3 steps (lines 44-69).

Since matrix dimensions are a multiple of 32, and element size is 8-bit, in the case of vector lengths higher than 256-bit, the vector loads of the last iteration of the inner loop might 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:

```
void rearrangeRight_fixp( uint64_t K, uint64_t N,
                          uint8_t * inRight, uint8_t * inRight_MOD) {
    uint64_t x, y, nb_st_elems;
    svbool_t p_ld;
    svuint8_t r0, r1, r2, r3;
    uint64_t offsetIN_1, offsetIN_2, offsetIN_3;
    uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3;

    uint8_t *ptrIN;
    uint8_t *ptrOUT;

    svbool_t p8_all = svptrue_b8();
    uint64_t vl = svcntb();
    uint64_t vl_4 = (vl >> 2);

    offsetIN_1 = N;
    offsetIN_2 = 2*N;
    offsetIN_3 = 3*N;

    offsetOUT_1 = K*vl_4;
    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 the right-matrix involves transposing multiple [4, vl/4] size sub-blocks (vl is the number of 8-bit elements in a machine vector length). This rearrangement is implemented with `ZIP1` and `ZIP2` instructions in 2 steps (lines 37-45).

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

The following is an SVE implementation of a matrix multiply results calculation:

```
void matmul_dotp_fixp( 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;
    svuint32_t acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
    uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3, offsetOUT_4;
    uint64_t offsetOUT_5, offsetOUT_6, offsetOUT_7;

    uint8_t *ptrIN_left;
    uint8_t *ptrIN_right;
    uint32_t *ptrOUT;

    svbool_t p8_all = svptrue_b8();
    uint64_t vl = svcntb();
    uint64_t vl_4 = (vl >> 2);

    offsetOUT_1 = N;
    offsetOUT_2 = 2*N;
    offsetOUT_3 = 3*N;
    offsetOUT_4 = 4*N;
    offsetOUT_5 = 5*N;
    offsetOUT_6 = 6*N;
    offsetOUT_7 = 7*N;

    for (x=0; x<M; x+=8) {
        ptrOUT = &out[x*N];
        ptrIN_right = &inRight_MOD[0];

        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 = svld1rq(p8_all, ptrIN_left);
42                   svuint8_t a4567 = svld1rq(p8_all, &ptrIN_left[16]);
43                   svuint8_t b_vec = svld1(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 innermost 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 fixed-point dot-product, by indexed elements `UDOT` instruction, at lines 45-52. The indexed elements are left-input matrix elements that are 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 innermost loop a subblock:

$$out[8, vl = 4]$$

results are stored to the memory by a contiguous vector length store `ST1W` (lines 58-65).

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

Related concepts

4.4.1 Matrix Multiplication with Real DP Floating-Point Elements on page 4-41

4.4.2 Matrix Multiplication with Real HP Floating-Point Elements on page 4-42

4.4.3 Matrix Multiplication with Real Fixed-Point 8-Bit Input Elements and Real Fixed-Point 32-Bit Output Elements on page 4-46

Chapter 5
Arm Instruction Emulator

Arm Instruction Emulator (ArmIE) runs on AArch64 platforms and emulates SVE instructions. ArmIE enables you to compile SVE code with Arm Compiler and run the SVE binary without SVE-enabled hardware. Based on the DynamoRIO dynamic binary instrumentation framework, ArmIE enables the customized instrumentation of SVE binaries allowing you to analyze specific aspects of runtime behavior.

It contains the following sections:

5.1 Introduction to Arm Instruction Emulator (ArmIE)

Arm Instruction Emulator (ArmIE) is a tool that converts instructions that are not supported on hardware to native Armv8-A instructions, such as those from the Scalable Vector Extension (SVE) instruction set.

ArmIE enables developers to run and test the *Scalable Vector Extension (SVE)* binaries on existing Armv8-A hardware, without resorting to simulators with high overheads. This approach trades-off performance accuracy (for example, ArmIE does not provide any timing information) for faster application execution time. Faster application execution time allows for larger, more realistic applications to be run, coupled with dynamic binary instrumentation.

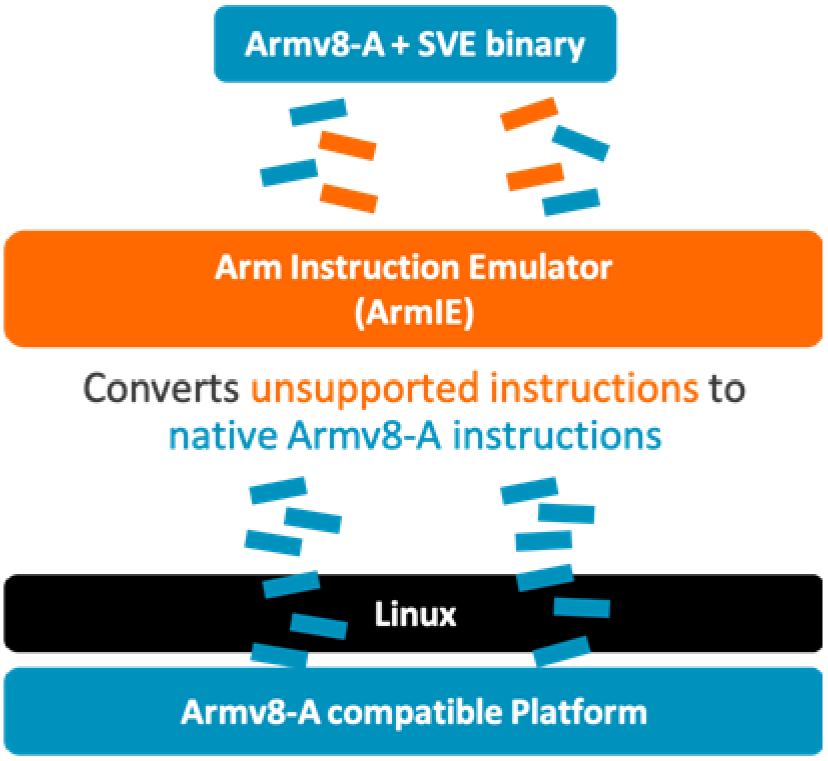
Armv8-A + SVE binary
Arm Instruction Emulator (ArmIE)
Converts unsupported instructions to native Armv8-A instructions
Linux
Armv8-A compatible Platform

Figure 5-1 ArmIE instruction flow diagram

Dynamic binary instrumentation support is provided through *DynamoRIO* integration, extending ArmIE capabilities beyond simple emulation. Instrumentation enables the collection of dynamic characteristics and metrics from the executing application, such as memory traces and instruction counts, allowing a deeper and more insightful analysis. Given the wide range of potential instrumentation which can be used and the metrics that can be gathered with ArmIE and DynamoRIO, it also includes *instrument emulated instructions* to the *DynamoRIO API*, allowing developers to build their own DynamoRIO clients with access to emulated instructions, when required. To help understand how the emulated instruction instrumentation functions of the API work, ArmIE includes four example instrumentation clients and their respective source codes, with emulation support. These clients are based on existing DynamoRIO ones and are:

- Instruction count client with emulated SVE (`samples/inscount_emulated.cpp`)
- Instruction count client (emulation API in the code, but no emulated SVE) (`samples/inscount.cpp`)
- Opcode count client (`samples/opcodes_emulated.cpp`)
- Memory tracing client (`samples/memtrace_simple.c`)

The structure adopted by ArmIE can be seen in the following diagram. Conceptually, ArmIE consists of an emulation client (currently for SVE) and optional instrumentation clients (for example, instruction count), which communicate between each other using the emulator API. Additional information on the clients, how ArmIE works, and how to set it up can be seen in the documentation provided with the tool and on the *ArmIE website*.

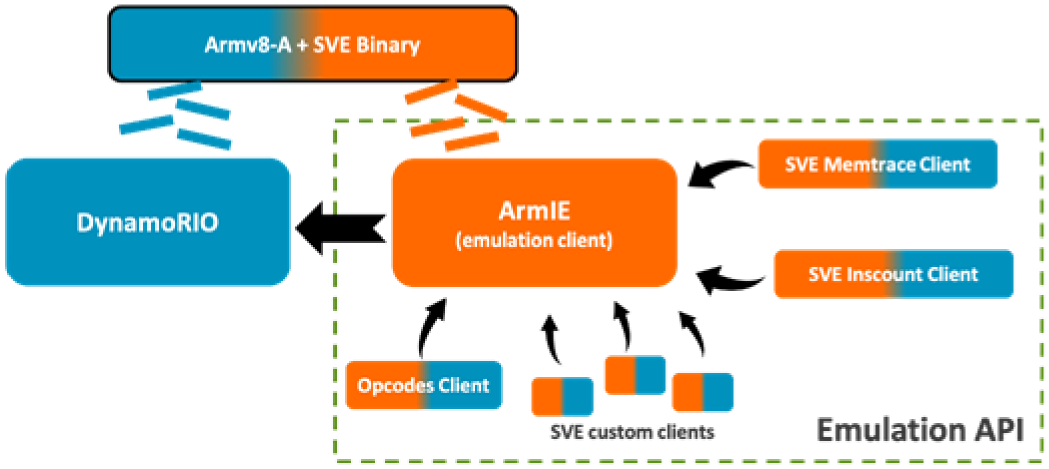

Figure 5-2 ArmIE and DynamoRIO interaction diagram

——— **Note** ———

ArmIE is not capable of producing timing information and incurs an emulation and binary instrumentation overhead on the running application. Therefore, no real-time performance considerations should be done based on these results.

———

5.2 Get Started

Demonstrates how to compile SVE code, run the resulting executable, and analyze runtime behavior with Arm Instruction Emulator.

Prerequisites

This tutorial also uses Arm Compiler for HPC. Ensure you have installed the Arm Compiler, and loaded the environment module before beginning this tutorial. See *Installing Arm Compiler for HPC* for installation and configuration instructions.

Note

The following instructions are relevant for Arm Instruction Emulator versions 18.0 and later. If you are using a previous version of Arm Instruction Emulator, please *download the Arm Instruction Emulator v1.2.1 user guide* instead of following the steps here.

Procedure

1. Install Arm Instruction Emulator.

   Refer to *Installing Arm Instruction Emulator* for details on how to perform the installation on Linux.
2. Configure your environment. To check which Environment Modules are available, enter:

   ```
   module avail
   ```

   Note

   You may need to configure the `MODULEPATH` environment variable to include the installation directory:

   ```
   export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
   ```

   ---

3. Load the Arm Instruction Emulator module to make it available for use:

   ```
   module load <architecture>/<linux_variant>/<linux_version>/suites/arm-instruction-
   emulator/<version>
   ```

   For example:

   ```
   module load Generic-AArch64/SUSE/12/suites/arm-instruction-emulator/18.4
   ```
4. Check your environment by examining the `PATH` variable. It should contain the appropriate Arm Instruction Emulator `bin` directory from `/opt/arm`:

   ```
   echo $PATH /opt/arm/arm-instruction-emulator-18.4_Generic-AArch64_SUSE-12_aarch64-linux/
   bin64:...
   ```
5. Run and instrument SVE binaries.

   To generate a SVE binary, you have to use SVE-capable compilers such as the *Arm HPC Compiler* or GCC 8.2+. You also need to enable the SVE architecture flag (for example, `-march=armv8-a+sve`).

Example 5-1 Examples

This example demonstrates how to compile and vectorize some C or Fortran code to target the SVE-enabled Armv8-A architecture, and how to emulate running the SVE code using Arm Instruction Emulator.

For the example provided, we use:

- Arm Compiler version 19.1
- ArmIE version 19.0

The example program subtracts corresponding elements in two arrays, and writes the result to a third array.

Next Steps

To learn about analyzing SVE code, see the next section *Analyze Emulated SVE on Existing Armv8-A Hardware* on page 5-56.

5.3 Analyze Emulated SVE on Existing Armv8-A Hardware

When developing high performance programs, runtime analysis is required to gain insights into execution behavior. Runtime analysis enables developers to identify heavily used loops and instruction sequences so that, for example, improvements can be made to execution speed and memory access.

ArmIE is based on the *DynamoRIO dynamic binary instrumentation tool platform* (DBI). ArmIE allows developers to use *DynamoRIO's API* to write instrumentation clients, which run alongside the SVE emulation client to analyze SVE binaries at runtime.

Before looking at an example of an instrumentation client for emulated binaries using ArmIE, Arm recommends you understand the basic principles of instrumenting binaries using the DynamoRIO API. See DynamoRIO's *API Usage Tutorial*.

Overview

In this topic, the *HACCKernels* mini-app which implements HACC's particle force kernels, is used as example code. The makefile changes to point to the Arm HPC compiler (`armclang++`) and adds the necessary SVE flag (`-march=armv8-a+sve`). To simplify the instrumentation output analysis, the OpenMP flag for these examples is removed.

In the source code, small modifications are made in the `main.cpp` file. To reduce the execution time for the evaluation presented here (~12x reduction), the number of iterations (`int NumIters`) is reduced from 2000 to 500, and runs only the 5th-order kernel (out of the available three kernels). Running only the 5th-order kernel provides a clear breakdown of the SVE impact in HACCKernels, both in instruction utilization and memory accesses. No other changes to the code are made at this point.

Instruction Count

Start with the instruction count client (inscount), choose a vector length of 512 bits. This client counts all the dynamic instructions that are executed by the binary, separating SVE instructions from AArch64 instructions. At this point in time, there is no breakdown on the types of instructions available in the client. In addition, the inscount client prints the emulated SVE instruction opcodes (and PC) to output. This can be decoded to obtain extra information about which instructions were executed. This is discussed in more detail in the next section.

```
armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./HACCKernels
Gravity Short-Range-Force Kernel (5th Order): 9178.27 -835.505 -167.99: 42.9214 s
205464290 instructions executed of which 167576110 were emulated instructions
```

From this inscount run, you see a very high number of emulated SVE instructions (81.56% of the total instructions), which demonstrates a good use of the vector extension.

The default mode of the inscount client counts all the executed instructions, including ones from shared libraries. To disable the count of shared libraries, you can enable a client flag, which leads to a higher SVE utilization rate of 83.00%. The example below demonstrates how to run the inscount client with this flag and its respective result. The run command for this case differs from the previous one because it is the DynamoRIO command (see *View the drrun command* on page 5-61) which ArmIE uses to load and run the emulation and instrumentation clients. This underlying DynamoRIO command can be exposed when running the ArmIE command using the `-s` option. In this case, the `-only_from_app` string is passed to the instrumentation client, `libinscount_emulated.so`, as a parameter to ignore all instruction counting except those in the application. `libsve_512.so` is the SVE emulation client.

```
$ARMIE_PATH/bin64/drrun -client $ARMIE_PATH/lib64/release/libsve_512.so 0 "" -client
$ARMIE_PATH/samples/bin64/libinscount_emulated.so 1 "-only_from_app" -max_bb_instrs 32 -
max_trace_bbs 4 -- ./HACCKernels
Gravity Short-Range-Force Kernel (5th Order): 9178.27 -835.505 -167.99: 42.7263 s
201887951 instructions executed of which 167576110 were emulated instructions
```

With the inscount client, you can also quickly compare the SVE utilization between different vector lengths. The table below shows the SVE utilization for vector lengths between 128 bits and 1024 bits.

Table 5-1 SVE Utilization (no shared libs) for different vector lengths

Vector length	128-bit	256-bit	512-bit	1024-bit
SVE utilization	93.43%	89.61%	83.00%	72.49%

The important conclusion from this table is that you can see that as total number of SVE instructions reduces, the wider the vectors get. This result is an expected occurrence because wider vectors can store more data and perform more simultaneous operations, reducing the total number of SVE instructions.

Opcodes Count

Similar to the inscount client, the opcodes client reports the dynamic count of the total number of instructions executed, separated by opcode. This client is useful for understanding the *hotness* factor of SVE instructions, and to correlate it against the source code of the application. Non-SVE opcodes are decoded by DynamoRIO, resulting in the corresponding mnemonics that can be seen in the output below:

```
Opcode execution counts in AArch64 mode:
      184763 : ubfm
      224217 : cbnz
      236845 : and
      253632 : ldrb
      481172 : adrp
      624493 : orr
      739335 : add
      810385 : fadd
     1017337 : subs
     1172879 : ldr
     1320770 : fmadd
     2792022 : xx
     3127984 : str
     4263314 : fcvt
     5342564 : bcond
     5833081 : fmul
     8473704 : eor
77 unique emulated instructions written to undecoded.txt
```

The unique SVE instruction opcodes are written to an output file (`undecoded.txt`) which can then be decoded. To facilitate this process, ArmIE includes a decoder script, `bin64/enc2instr.py`, that uses the *LLVM machine code* (llvm-mc) binary (available in the Arm Compiler), to disassemble the instruction encodings. Using this script, you can obtain a breakdown of the SVE instructions, with their mnemonics and accessed registers, as shown below.

Note

The provided script is written for a generic case where a single encoding can be passed to it, and not specifically for this client. When running the script, you need to remove the instruction count, present in the `undecoded.txt` file, to avoid any incompatibilities. The single command line shown below extracts the encodings from the generated file, runs them through the script, and pastes back together the instruction count with the respective decoding, all in a single command line:

```
awk '{print $3}' undecoded.txt | $ARMIE_PATH/bin64/enc2instr.py -mattr=+sve | awk -F:
'{print $2}' | paste undecoded.txt /dev/stdin
 4150900 : 0xa5484c9b         ld1w   {z27.s}, p3/z, [x4, x8, lsl #2]
 4150900 : 0xa5484479         ld1w   {z25.s}, p1/z, [x3, x8, lsl #2]
 4150900 : 0xa5484458         ld1w   {z24.s}, p1/z, [x2, x8, lsl #2]
 4150900 : 0xa5484437         ld1w   {z23.s}, p1/z, [x1, x8, lsl #2]
 4150900 : 0x65b9033a         fmla   z26.s, p0/m, z25.s, z25.s
 4150900 : 0x65b8031b         fmla   z27.s, p0/m, z24.s, z24.s
 4150900 : 0x65b68359         fmad   z25.s, p0/m, z26.s, z22.s
 4150900 : 0x65b58358         fmad   z24.s, p0/m, z26.s, z21.s
 4150900 : 0x65b4837a         fmad   z26.s, p0/m, z27.s, z20.s
 4150900 : 0x65b3e35b         fnmsb  z27.s, p0/m, z26.s, z19.s
 4150900 : 0x65b2e35b         fnmsb  z27.s, p0/m, z26.s, z18.s
 4150900 : 0x65b1e35b         fnmsb  z27.s, p0/m, z26.s, z17.s
 4150900 : 0x65a6635b         fnmls  z27.s, p0/m, z26.s, z6.s
 4150900 : 0x65a58357         fmad   z23.s, p0/m, z26.s, z5.s
 4150900 : 0x659c0bbc         fmul   z28.s, z29.s, z28.s
 4150900 : 0x659c035a         fadd   z26.s, z26.s, z28.s
 4150900 : 0x659b0b5a         fmul   z26.s, z26.s, z27.s
 4150900 : 0x65970afa         fmul   z26.s, z23.s, z23.s
```

```
4150900 : 0x65922343         fcmeq  p3.s, p0/z, z26.s, #0.0
4150900 : 0x658da39d         fsqrt  z29.s, p0/m, z28.s
4150900 : 0x658c80fc         fdivr  z28.s, p0/m, z28.s, z7.s
4150900 : 0x6584035c         fadd   z28.s, z26.s, z4.s
4150900 : 0x65834342         fcmge  p2.s, p0/z, z26.s, z3.s
4150900 : 0x65820739         fsub   z25.s, z25.s, z2.s
4150900 : 0x65810718         fsub   z24.s, z24.s, z1.s
4150900 : 0x658006f7         fsub   z23.s, z23.s, z0.s
4150900 : 0x25a91d01         whilelo        p1.s, x8, x9
4150900 : 0x25834042         orr    p2.b, p0/z, p2.b, p3.b
4150900 : 0x25034023         and    p3.b, p0/z, p1.b, p3.b
4150900 : 0x25004243         not    p3.b, p0/z, p2.b
4150900 : 0x05b9cad9         mov    z25.s, p2/m, z22.s
4150900 : 0x05b8cab8         mov    z24.s, p2/m, z21.s
4150900 : 0x05b7c8b7         mov    z23.s, p2/m, z5.s
4150900 : 0x05b6c736         mov    z22.s, p1/m, z25.s
4150900 : 0x05b5c715         mov    z21.s, p1/m, z24.s
4150900 : 0x05a5c6e5         mov    z5.s, p1/m, z23.s
4150900 : 0x04b0e3e8         incw   x8
4150900 : 0x0420bf7a         movprfx        z26, z27
4150900 : 0x0420bf5b         movprfx        z27, z26
4150900 : 0x0420be1b         movprfx        z27, z16
 166036 : 0x25351d00         whilelo        p0.b, x8, x21
 166036 : 0x252c8808         incp   x8, p0.b
  83018 : 0xe4084000         st1b   {z0.b}, p0, [x0, x8]
  50000 : 0x658022c0         faddv  s0, p0, z22.s
  50000 : 0x658022a1         faddv  s1, p0, z21.s
  50000 : 0x658020a2         faddv  s2, p0, z5.s
  50000 : 0x25b9ce07         fmov   z7.s, #1.00000000
  50000 : 0x25b8c005         mov    z5.s, #0                // =0x0
  50000 : 0x25a91fe1         whilelo        p1.s, xzr, x9
  50000 : 0x2598e3e0         ptrue  p0.s
  50000 : 0x05242294         mov    z20.s, s20
  50000 : 0x05242273         mov    z19.s, s19
  50000 : 0x05242252         mov    z18.s, s18
  50000 : 0x05242231         mov    z17.s, s17
  50000 : 0x05242210         mov    z16.s, s16
  50000 : 0x052420a6         mov    z6.s, s5
  50000 : 0x05242084         mov    z4.s, s4
  50000 : 0x05242063         mov    z3.s, s3
  50000 : 0x05242042         mov    z2.s, s2
  50000 : 0x05242021         mov    z1.s, s1
  50000 : 0x05242000         mov    z0.s, s0
  50000 : 0x046530b6         mov    z22.d, z5.d
  50000 : 0x046530b5         mov    z21.d, z5.d
  41509 : 0xe4084380         st1b   {z0.b}, p0, [x28, x8]
  41509 : 0xe40842c0         st1b   {z0.b}, p0, [x22, x8]
  10500 : 0x253c1d00         whilelo        p0.b, x8, x28
  10500 : 0x04285028         addvl  x8, x8, #1
   3500 : 0xe40842e0         st1b   {z0.b}, p0, [x23, x8]
   3500 : 0xe4084280         st1b   {z0.b}, p0, [x20, x8]
   3500 : 0xe4084260         st1b   {z0.b}, p0, [x19, x8]
   3500 : 0x2538c000         mov    z0.b, #0                // =0x0
   3000 : 0x04bf5028         rdvl   x8, #1
   2000 : 0x25351fe0         whilelo        p0.b, xzr, x21
   1500 : 0x253c1fe0         whilelo        p0.b, xzr, x28
    500 : 0x04bf5029         rdvl   x9, #1
```

Memory Tracing

The memory tracing client (memtrace) focuses on the dynamic memory accesses of the application, capturing information such as the accessed addresses and data sizes. Memtrace is based on the existing non-SVE DynamoRIO memtrace client, with added SVE emulation and tracing support. Running the emulated memtrace client results in two different memory trace files: an SVE-only trace and a non-SVE one. To keep the memory traces consistent, include an additional field, ‘Sequence Number’, that updates the order of each memory access sequentially, through a shared counter between the emulation side and the core DynamoRIO instrumentation. The memory trace format is the following:

- Sequence Number
- Thread ID
- SVE Bundle
- isWrite (1 = write, 0 = read)
- Data Size (Bytes)
- Data Address
- PC

ArmIE also adds the ‘SVE Bundle’ field to the memory traces, which identifies SVE linear and gather/scatter vector accesses. It consists of 3 bits with the following possible combinations appearing in the resulting trace:

- 0: Contiguous access
- 1 or 3: Gather/Scatter bundle, first element
- 2: Gather/Scatter bundle, another element
- 4 or 6: Gather/Scatter bundle, last element

An important consideration to have when tracing SVE binaries is that the output trace can easily use up a large amount of disk space. Therefore, ArmIE supports marker instructions that you must include in your SVE code to define start and end regions (multiple regions are supported) where the memtrace client will execute. In a typical scenario, this corresponds to the main kernel loops of the application.

Note

Only the region inside these markers is traced. If they are not used, no tracing is done. These markers should also be outside vectorizable loops, as they might hinder vectorization.

In the case of the HACCKernels mini-app example, add the marker definitions at the start of the `main.cpp` file and define the region of interest around the main kernel, GravityForceKernel5:

```
#define __START_TRACE() {asm volatile (".inst 0x2520e020");}
#define __STOP_TRACE() {asm volatile (".inst 0x2520e040");}
....
__START_TRACE();
run(GravityForceKernel5, "5th Order");
__STOP_TRACE();
#define __START_TRACE() {asm volatile (".inst 0x2520e020");}
#define __STOP_TRACE() {asm volatile (".inst 0x2520e040");}
....
__START_TRACE();
run(GravityForceKernel5, "5th Order");
__STOP_TRACE();
```

After you add the region markers to the code, compile it, and run it with the memtrace client (again with 512-bit vectors):

```
armie -e libmemtrace_sve_512.so -i libmemtrace_simple.so -- ./HACCKernels
> Data file /home/migtai01/apps-unimplemented/HACCKernels_sve_vectorizer/
memtrace.HACCKernels.03531.0000.log created
> Gravity Short-Range-Force Kernel (5th Order): 9178.27 -835.505 -167.99: 73.5272 s
ls
> memtrace.HACCKernels.0000.log
> sve-memtrace.HACCKernels.8114.log
```

The combined trace files form over 22M of total trace lines, so only a small snippet is reported in this example. For analysis purposes, it is advantageous to merge both the non-SVE and SVE trace files into a single file. Merging files can be done with a simple script that parses the separate memory trace files and orders them into a single full trace output using on the ‘Sequence Number’ trace field. ArmIE does not currently include an example script for this function.

To facilitate analysis of the fully-merged memory trace, we use different separator characters after the first element of each trace line: a colon ‘ : ‘ separator for non-SVE traces, and a comma ‘ , ‘ separator for SVE traces. This is shown below, in the merged memory trace snippet of HACCKernels:

```
Format: <sequence number>: <TID>, <isBundle>, <isWrite>, <data size>, <data address>, <PC>
....
2990: 0, 0,  0,  4, 0x401a68, 0x401678
2991: 0, 0,  0,  4, 0x401a6c, 0x401680
2992: 0, 0,  0,  4, 0x401a70, 0x401688
2993: 0, 0,  0,  4, 0x401a74, 0x401690
2994: 0, 0,  0,  4, 0x401a78, 0x401698
2995: 0, 0,  0,  4, 0x401a7c, 0x4016a0
2996, 0, 0, 0, 64, 0x44c750, 0x4016f0
2997, 0, 0, 0, 64, 0x44d110, 0x4016f4
2998, 0, 0, 0, 64, 0x44dad0, 0x4016f8
2999, 0, 0, 0, 4, 0x44e490, 0x401750
3000, 0, 0, 0, 16, 0x44e49c, 0x401750
```

```
3001, 0, 0, 0, 4, 0x44e4b0, 0x401750
....
```

The memory trace snippet shows an initial section with non-SVE memory accesses (traces 2990 to 2995), followed by SVE accesses (traces 2996 to 3001). The accesses can be seen by the first separator character, after the sequence number. Only load accesses are captured in this memory trace snippet, but write operations are present in the full memory trace. Looking at the *size* field, you can also observe three full SVE-vector loads (64 byte size equals to 512-bit vector lengths).

Memory traces are commonly used for different types of post-processing analysis. This can encompass a wide-range of scripts and tools, ranging from simple parsing scripts to more complex cache simulators, and so on. Processing memory traces falls outside the scope of ArmIE and, therefore, no extra tools are currently included.

Below we present some simple scripting experiments that can be done to the fully-merged memory traces. It parses all memory traces and prints related information, for example, number of linear and gather/scatter bundle accesses, percentage of writes and reads, or accesses with inactive vector lanes. You can observe that the HACCKernels mini-application does not present a single gather/scatter operation and that load operations dominate the memory accesses performed.

```
SVE vector size: 512 bits (64B)
Total Memory References = 20486551
   -> linear SVE: 16779970 (81.91%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 3706581 (18.09%)
Linear SVE accesses with at least 1 inactive lane:
   -> 1237914 (7.38% of linear SVE traces)
==============
Total Writes = 3121089  (34 unique writes - different PCs)
   -> linear SVE: 176536 (5.66%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 2944553 (94.34%)
Linear SVE Writes with at least 1 inactive lane:
   -> 3376 (1.91% of linear SVE write traces)
==============
Total Loads = 17365462 (51 unique loads - different PCs)
   -> linear SVE: 16603434 (95.61%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 762028 (4.39%)
Linear SVE Loads with at least 1 inactive lane:
> 1234538 (7.44% of linear SVE loads)
==============
Distribution of memory operations:
   -> 15.23% Writes
   -> 84.77% Loads
SVE Bundles Stats:
   -> 0 SVE bundle accesses
```

Additional Resources

This topic is formed of extracts from a blog. To read the full blog, see *Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE*.

Summary

In this topic, an overview of the Arm Instruction Emulator, from its structure to the existing emulation clients and how to use them, is given. Types of analysis and studies that can be achieved with the provided clients, for example, with regards to memory traces and post-processing analysis. More complex post-processing analysis can be done on memory traces, such as using cache simulation, because the existent DynamoRIO cache simulator shows for different types of non-SVE traces. Although not yet compatible with ArmIE, it serves as an example of the type of analysis you can explore with the traces and metrics that you can gather.

5.4 View the drrun command

This example uses the `-s` or `--show-drrun-cmd` option when running ArmIE on a binary to output the full DynamoRIO drrun command that ArmIE uses. The `-s` option is provided to enable the full range of options for drrun, and to pass command-line arguments to clients. Without this feature, options and arguments need to be passed via the `armie` command.

Prerequisites

To gain an understanding of how ArmIE is used, work through the online *Get Started* and *Analyze SVE programs* tutorials.

Procedure

1. Run ArmIE with the `-s` option, using the example described in Get Started:

   ```
   armie -msve-vector-bits=128 -s -- ./example
   ```

   This returns:

   ```
   /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/bin64/
   drrun -max_bb_instrs 32 -max_trace_bbs 4 -c /path/to/your/arm-instruction-emulator-
   <xx.y>_Generic-AArch64_<OS>_aarch64-linux/lib64/release/libsve_128.so -- ./example
    i       a[i]    b[i]    c[i]
   =============================
   0       197     283     86
   1       262     277     15
   . . .
   1021    165     234     69
   1022    232     295     63
   1023    204     235     31
   ```

   Notice that drrun uses the emulation client libsve_128.so to run the example binary.

2. If an instrumentation client is specified:

   ```
   armie -msve-vector-bits=128 -s -i libinscount_emulated.so -- ./example
   ```

   This returns:

   ```
   /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/bin64/
   drrun -client /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-
   linux/lib64/release/libsve_128.so 0 "" -client /path/to/your/arm-instruction-emulator-
   <xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin64/libinscount_emulated.so 1 "" -
   max_bb_instrs 32 -max_trace_bbs 4 -- ./example
       Client inscount is running
       . . .
       1022    232     295     63
       1023    204     235     31
       2134094 instructions executed of which 1537 were emulated instructions
   ```

   Notice that drrun now uses two clients: the emulation client libsve_128.so and libinscount_emulated.so to run and count instructions executed by example.

3. The libinscount_emulated.so client has an option `-only_from_app` which only counts instructions executed by the application, rather than including linked libraries in addition. This enables users to copy and paste the above command adding `-only_from_app`:

   ```
   /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-linux/bin64/
   drrun -client /path/to/your/arm-instruction-emulator-<xx.y>_Generic-AArch64_<OS>_aarch64-
   linux/lib64/release/libsve_128.so 0 "" -client /path/to/your/arm-instruction-emulator-
   <xx.y>_Generic-AArch64_<OS>_aarch64-linux/samples/bin64/libinscount_emulated.so 1 "-
   only_from_app" -max_bb_instrs 32 -max_trace_bbs 4 -- ./example
       Client inscount is running
       . . .
       1021    165     234     69
       1022    232     295     63
       1023    204     235     31
       42902 instructions executed of which 1537 were emulated instructions
   ```

   Notice that the native AArch64 instruction count has dropped to 42902 from 2134094 due to the exclusion of library instructions.