Porting and Optimizing HPC Applications for Arm® SVE

Version 2.0

arm
Porting and Optimizing HPC Applications for Arm® SVE

Copyright © 2019 Arm Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100-00</td>
<td>17 May 2019</td>
<td>Non-Confidential</td>
<td>First release.</td>
</tr>
<tr>
<td>0110-00</td>
<td>05 June 2019</td>
<td>Non-Confidential</td>
<td>Document update to version 1.1.</td>
</tr>
<tr>
<td>0200-00</td>
<td>11 November 2019</td>
<td>Non-Confidential</td>
<td>Document update to version 2.0.</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

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


110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349
Confidentiality Status
This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

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

Web Address
www.arm.com
## Chapter 5  
**Arm Instruction Emulator**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction to Arm instruction Emulator (armie)</td>
<td>5-77</td>
</tr>
<tr>
<td>5.2</td>
<td>Get Started</td>
<td>5-79</td>
</tr>
<tr>
<td>5.3</td>
<td>Analyze Emulated SVE on Existing Arm®v8-A Hardware</td>
<td>5-81</td>
</tr>
<tr>
<td>5.4</td>
<td>View the drrun command</td>
<td>5-86</td>
</tr>
</tbody>
</table>
### List of Figures

Porting and Optimizing HPC Applications for Arm® SVE

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2-1</td>
<td>Register overlapping.</td>
<td>2-15</td>
</tr>
<tr>
<td>Figure 2-2</td>
<td>Matrix multiplication diagram</td>
<td>2-25</td>
</tr>
<tr>
<td>Figure 2-3</td>
<td>SVE optimization methodology</td>
<td>2-36</td>
</tr>
<tr>
<td>Figure 2-4</td>
<td>Compiler autovectorization comparison</td>
<td>2-37</td>
</tr>
<tr>
<td>Figure 2-5</td>
<td>Instruction count reduction when increasing vector length</td>
<td>2-38</td>
</tr>
<tr>
<td>Figure 2-6</td>
<td>Percentage of SVE instructions per computational kernel</td>
<td>2-39</td>
</tr>
<tr>
<td>Figure 2-7</td>
<td>Average lane utilization</td>
<td>2-40</td>
</tr>
<tr>
<td>Figure 2-8</td>
<td>Memory accesses breakdown</td>
<td>2-41</td>
</tr>
<tr>
<td>Figure 2-9</td>
<td>Simulator parameters</td>
<td>2-42</td>
</tr>
<tr>
<td>Figure 2-10</td>
<td>Average number of cycles per memory access</td>
<td>2-42</td>
</tr>
<tr>
<td>Figure 2-11</td>
<td>Optimization summary</td>
<td>2-43</td>
</tr>
<tr>
<td>Figure 5-1</td>
<td>ArmIE instruction flow diagram</td>
<td>5-77</td>
</tr>
<tr>
<td>Figure 5-2</td>
<td>ArmIE and DynamoRIO interaction diagram</td>
<td>5-78</td>
</tr>
</tbody>
</table>
List of Tables
Porting and Optimizing HPC Applications for Arm® SVE

Table 2-1 Summary of the Instruction Set extensions ................................................................. 2-21
Table 2-2 Auto-vectorization with Arm Compiler for Linux ...................................................... 2-23
Table 2-3 Supported Arm Compiler for Linux optimization options for auto-vectorization .......... 2-23
Table 2-4 Summary of Neon intrinsics used ............................................................................. 2-26
Table 2-5 Summary of SVE intrinsics used .............................................................................. 2-28
Table 2-6 Neon and SVE Macros ............................................................................................. 2-31
Table 2-7 Neon and SVE Types .............................................................................................. 2-32
Table 5-1 SVE Utilization (no shared libs) for different vector lengths ...................................... 5-82
Preface

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

It contains the following:
About this book

Guide to porting and optimizing High Performance Computing (HPC) applications for Arm® Scalable Vector Extension (SVE).

Using this book

This book is organized into the following chapters:

Chapter 1 Porting to Arm Resources
This guide supplements the other resources which Arm provides to help you start 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 is best for the workloads that 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 some SVE programming tips that are supported by assembly examples 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 for Linux 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
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_0200_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 which Arm provides to help you start 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

To help you port your applications to AArch64, Arm provides a variety of resources:

- **Porting and Optimizing HPC Applications for Arm** guide. This guide discusses the tools Arm offers to help you port your codes, includes some step-by-step information about how to approach a code porting task, provides a compiler comparison for new Arm Compiler for Linux users, and also includes some useful information about coding for Neon™.
- **Arm-supported porting and tuning recipes**. This page contains a list of Arm-supported recipes to port common applications, benchmarks, and libraries to AArch64.
- Community-driven porting recipes on the Arm HPC Packages Wiki. The GitLab repository contains a broader list of porting and tuning recipes to which the community contribute and maintain.
- **White papers**. A collection of White Papers and externally-published papers that are relevant to High Performance Computing (HPC) on Arm.
- **Conference presentations**. Arm frequently presents and hosts events at HPC industry conferences. This page collects the presentations delivered at these events.
- **Arm Allinea Studio documentation**. Arm Allinea Studio is composed of Arm Fortran Compiler, Arm C/C++ Compiler, Arm Performance Libraries, Arm Forge, and Arm Performance Reports. More information about the tools is available on the Arm Allinea Studio page.
- **HPC blogs and forums on Arm HPC Community**. Arm frequently publishes information about new releases, interesting research, conference reviews, and industry news through blogs. The forum is available for the Arm HPC community to ask questions and find solutions in a forum context.
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 is best for the workloads that 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 and unsigned extended loads.

It contains the following sections:

• 2.1 What is the Scalable Vector Extension? on page 2-14.
• 2.2 Why is the Scalable Vector Extension (SVE) useful for HPC? on page 2-19.
• 2.3 Coding for SVE vs Neon™ on page 2-20.
• 2.4 Case Study: Optimizing HPCG for Arm SVE on page 2-35.
• 2.5 Additional resources on page 2-44.
2.1 What is the Scalable Vector Extension?

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

Introduction

Scalable Vector Extension (SVE) is the next-generation SIMD extension of the Armv8-A AArch64 instruction set. SVE is not an extension of Neon, but a new set of vector instructions that are 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 (VLA). SVE does not fix the size of the vector registers which allows hardware implementors to choose the size that is best for their 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 use condition-creating instructions, such as compares. Condition-creating instructions allow later instructions to control which elements (or ‘lanes’) that a vector should be operated on (the ‘active’ elements).

Gather-load and scatter-store

Gather-load and scatter-store allows data to be efficiently transferred to or from a vector of non-contiguous memory addresses. The efficient transfer of data enables a wider range of source code constructs to be vectorized. To permit efficient accesses to contiguous memory, SVE provides an extensive 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

Fault-tolerant speculative vectorization suppresses memory faults if they do not occur because of the first active element of the vector. Instead, fault-tolerant speculative vectorization generates a predicate value indicating which of the requested lanes were successfully loaded prior to the first memory fault. This indication 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 bit-wise logical reductions.
Serialized vector operations

SVE allows you to perform serial pointer-chasing loops and to use a vector of addresses with an associated predicate, allowing you to parallelize the remainder of the loop.

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. 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.

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 register provides 128-bit predicate registers.

The FFR register is a ‘special’ predicate register that differs from regular predicate registers because it is 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 are 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:

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 various 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 that operate on data registers and produce predicate registers (for example, numeric comparisons). * ‘predicate manipulation’ instructions, that include predicate generation or logical operations on predicates.

Note

You can only use the predicate registers p0 through p7 as predicates in data-processing instructions.

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

A large 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. Vector load/store instructions also have a wide range of new addressing modes that improve the efficiency of SVE code.

SVE instructions can be separated by function:

Note

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 the traditional strengths of Neon 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.
- SVE lowers 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. Using the VLA programming model allows you to compile or hand-code your program for SVE hardware once, and avoid the need to recompile or rewrite it when longer vectors appear in the future. This coding approach reduces deployment costs over the lifetime of the architecture; a program ‘works everywhere’ and executes wider and faster. Importantly, this coding approach 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. If a wider vector unit is available, a compiler can vectorize this code and make good use of the wider vector unit. To the benefit of SVE, HPC clusters are built with the wide, high-bandwidth memory systems that are necessary to feed a longer vector unit. In addition, the SVE instructions are designed to support full floating-point features that comply with the IEEE 754 standard, including half-precision floating-point.

Related concepts
2.1 What is the Scalable Vector Extension? on page 2-14

Related information
New architectural features
2.3 Coding for SVE vs Neon™

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

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

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

Data processing methodologies: SISD and SIMD

Most Arm instructions are Single Instruction Single Data (SISD). Each instruction performs one operation and writes to one output data stream. Processing multiple items requires multiple instructions.

For example, in traditional SISD instruction sets, to perform four separate addition operations requires four instructions to add values from four pairs of registers:

```
ADD x0, x0, x5
ADD x1, x1, x6
ADD x2, x2, x7
ADD x3, x3, x8
```

Single Instruction Multiple Data (SIMD) instructions perform the same operation simultaneously for multiple items. These items are packed as separate elements in a larger register.

For example, the following instruction adds four pairs of single-precision (32-bit) values together. However, in this case, the values are packed as separate lanes in one pair of 128-bit registers. Each lane in the first source register is then added to the corresponding lane in the second source register, before being stored in the destination register:

```
ADD Q8.4S, Q8.4S, Q9.4S
```

Performing the four operations with a single SIMD instruction is more efficient than with four separate SISD instructions.

Fundamentals: Instruction sets

AArch64 is the name that is used to describe the 64-bit Execution state of the Armv8 architecture. In AArch64 state, the processor executes the A64 Instruction Set, which contains Neon instructions (also referred to as Advanced SIMD instructions). The SVE extension is introduced in version Armv8.2-A of the architecture, and adds a new subset of instructions to the existing Armv8-A A64 Instruction Set.
### Table 2-1  Summary of the Instruction Set extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Key feature(s)</th>
<th>Categorization of new instructions</th>
</tr>
</thead>
</table>
| Neon      | Provides additional instructions that can perform mathematical operations in parallel on multiple data streams. Support for double precision floating-point, enabling C code using double precision. | • Promotion/Demotion  
• Pair-wise operations  
• Load and store operations  
• Logical operators  
• Multiplication operation |
| SVE       | SVE adds: * Support for wide vector and predicate registers (resulting in two main classes of instructions; **predicated** and **unpredicated**). * A set of instructions that operate on wide vectors. * Some minor additions to the configuration and identification registers. | • Load, store, and prefetch instructions.  
• Integer operations.  
• Vector address calculation.  
• Bitwise operations.  
• Floating-point operations.  
• Predicate operations.  
• Move operations.  
• Reduction operations.  
For descriptions of each, see *What is the Scalable Vector Extension?* on page 2-14. |

For more information about the Neon instruction set, see the *A64 Instruction set for Armv8-A*. For more information about the SVE instruction set extension, see *ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*.

### Fundamentals: Registers, vectors, lanes, and elements

Neon units operate on a separate register file of 128-bit registers and are fully integrated into Armv8-A processors. Neon units use a simple programming model because they use the same address space as an application.

The Neon register file is a collection of registers which can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit registers.

The Neon registers contain **vectors** of some consistent data type. A vector is divided into **lanes** and each lane contains a data value that is called an **element**.

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

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

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

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

- The **Z** registers are data registers. **Z** register bits are an implementation defined multiple of 128, up to an architectural maximum of up to 2048-bits. Data in these registers can be interpreted as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit. The low 128 bits of each **Z** register overlap the corresponding Neon registers, and therefore also the scalar floating-point registers.
- In every processor implementation, the **P** registers hold one bit for each byte available in a **Z** register. In other words, a **P** register is always 1/8th the size of the **Z** register width.
- The FFR register is a special predicate register that certain instructions can use implicitly. Both **P** registers and the FFR register are unique to SVE.
Fundamentals: Vector Length Agnostic (VLA) programming

SVE introduces the concept of 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 freedom of choice 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, so it includes instructions which permit vector code to adapt automatically to the current vector length at runtime.

More information about VLA programming is provided in a later chapter, see SVE Vector Length Agnostic programming on page 3-45.

Coding best practices

As a programmer, there are a number of ways you can make use of Neon and SVE technology:

• Neon and SVE-enabled math libraries, such as Arm Performance Libraries.

  Note SVE-enabled library introduced in Arm Compiler for Linux version 19.3+.

• Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Neon and SVE.
• Intrinsics are function calls that the compiler replaces with appropriate Neon or SVE instructions. This gives you direct access to the exact Neon or SVE instructions you want. For a searchable index for Neon intrinsics, see Neon intrinsics. The SVE intrinsics are defined in the Arm C Language Extensions for SVE specification.
• For very high performance, hand-coded Neon or SVE assembly code can be an alternative approach for experienced programmers.

Coding best practices: Compiler optimization

The Arm Compiler for Linux can automatically generate code that contains Armv8 Neon and SVE instructions. Allowing the compiler to automatically identify opportunities in your code to use Neon or SVE instructions is called auto-vectorization.

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

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

The benefits of relying on compiler auto-vectorization are:

• Programs implemented in high level languages are portable, so long as there are no architecture-specific code elements such as inline assembly or intrinsics.
• Modern compilers are capable of performing advanced optimizations automatically.
• Targeting a given micro-architecture can be as easy as setting a single compiler option. However, hand-optimizing a program in assembly requires deep knowledge of the target hardware.

To compile for AArch64 with Arm Compiler for Linux, see the following quick reference table:
### Table 2-2  Auto-vectorizaton with Arm Compiler for Linux

<table>
<thead>
<tr>
<th>Extension</th>
<th>Header file form</th>
<th>Recommended Arm Compiler for Linux command line</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neon</td>
<td>#include &lt;arm_neon.h&gt;</td>
<td>armclang -O&lt;level&gt; -mcpu={native</td>
<td>&lt;target&gt;} -o &lt;binary_name&gt; &lt;filename&gt;.c</td>
</tr>
<tr>
<td>SVE</td>
<td>#ifdef __ARM_FEATURE_SVE ifndef __ARM_FEATURE_SVE */</td>
<td>armclang -O&lt;level&gt; -march=armv8-a+sve -o &lt;binary_name&gt; &lt;filename&gt;.c</td>
<td><code>-march=armv8-a+sve</code> ensures the compiler optimizes for Armv8-A hardware, on which you can use ArmIE to emulate the SVE instructions. When SVE-enabled hardware is available and you are compiling on that target SVE hardware, Arm recommends using <code>-mcpu=native</code> instead, so that micro-architectural optimizations can be taken advantage of.</td>
</tr>
</tbody>
</table>

Supported optimization levels for `-O<level>` for both Neon and SVE code include:

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

• Auto-vectorization is enabled by default at optimization level -O2 and higher. The -fno-vectorize option lets you disable auto-vectorization.
• At optimization level -O0, auto-vectorization is always disabled. If you specify the -fvectorize option, the compiler ignores it.
• At optimization level -O1, auto-vectorization is disabled by default. The -fvectorize option lets you enable auto-vectorization.

As an implementation becomes more complicated, the likelihood that the compiler can auto-vectorize the code decreases. For example, loops with the following characteristics are particularly difficult (or impossible) to vectorize:
• Loops with interdependencies between different loop iterations.
• Loops with break clauses.
• Loops with complex conditions.

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

Note

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

A full discussion of the compiler directives used to control vectorization of loops for can be found in the LLVM-Clang documentation, but the two most important are:
• #pragma clang loop vectorize(enable)
• #pragma clang loop interleave(enable)

These pragmas are hints to the compiler to perform SLP and Loop vectorization respectively. More detailed guides covering auto-vectorization are available in the Arm C/C++ Compiler and Arm Fortran Compiler Reference guides:

Coding best practices: Intrinsics

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

Using intrinsics has several benefits:
• Powerful: Intrinsics give the developer direct access to the Neon and SVE instruction sets, without the need for hand-written assembly code.
• Portable: Hand-written Neon or SVE assembly instructions might need to be rewritten for different target processors. C and C++ code containing Neon intrinsics can be compiled for a new AArch64 target or a new Execution state with minimal or no code changes. However, C and C++ code containing SVE intrinsics will only run on SVE-enabled hardware.
• Flexible: The developer can exploit Neon when needed, or use C/C++ when it is not, while avoiding many low-level engineering concerns.

However, intrinsics might not be the right choice in all situations:
• More learning is required to use intrinsics, than to import a library or to rely on a compiler.
• Hand-optimized assembly code might offer the greatest scope for performance improvement, even if it is more difficult to write.

For a list of all the Neon intrinsics, see the Neon intrinsics. The Neon intrinsics engineering specification is contained in the Arm C Language Extensions (ACLE).

The SVE intrinsics engineering specification is contained in the Arm C Language Extensions for SVE specification.

**Example 1: Simple matrix multiplication with intrinsics**

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

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

The matrix multiplication process is as follows:

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

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

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

Assume a column-major layout of the matrices in memory. That is, an n x m matrix M, is represented as an array M_array, where \( M_{ij} = M\_array[n*j + i] \).

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

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

```c
void matrix_multiply_4x4_neon(const float32_t *A, const float32_t *B, float32_t *C) {
    // these are the columns A
```

![Figure 2-2 Matrix multiplication diagram](image-url)
float32x4_t A0;
float32x4_t A1;
float32x4_t A2;
float32x4_t A3;
// these are the columns B
float32x4_t B0;
float32x4_t B1;
float32x4_t B2;
float32x4_t B3;
// these are the columns C
float32x4_t C0;
float32x4_t C1;
float32x4_t C2;
float32x4_t C3;
A0 = vld1q_f32(A);
A1 = vld1q_f32(A+4);
A2 = vld1q_f32(A+8);
A3 = vld1q_f32(A+12);
// Zero accumulators for C values
C0 = vmovq_n_f32(0);
C1 = vmovq_n_f32(0);
C2 = vmovq_n_f32(0);
C3 = vmovq_n_f32(0);
// Multiply accumulate in 4x1 blocks, that is each column in C
B0 = vld1q_f32(B);
C0 = vfmq_laneq_f32(C0, A0, B0, 0);
C0 = vfmq_laneq_f32(C0, A1, B0, 1);
C0 = vfmq_laneq_f32(C0, A2, B0, 2);
C0 = vfmq_laneq_f32(C0, A3, B0, 3);
vstlq_f32(C, C0);
B1 = vld1q_f32(B+4);
C1 = vfmq_laneq_f32(C1, A0, B1, 0);
C1 = vfmq_laneq_f32(C1, A1, B1, 1);
C1 = vfmq_laneq_f32(C1, A2, B1, 2);
C1 = vfmq_laneq_f32(C1, A3, B1, 3);
vstlq_f32(C+4, C1);
B2 = vld1q_f32(B+8);
C2 = vfmq_laneq_f32(C2, A0, B2, 0);
C2 = vfmq_laneq_f32(C2, A1, B2, 1);
C2 = vfmq_laneq_f32(C2, A2, B2, 2);
C2 = vfmq_laneq_f32(C2, A3, B2, 3);
vstlq_f32(C+8, C2);
B3 = vld1q_f32(B+12);
C3 = vfmq_laneq_f32(C3, A0, B3, 0);
C3 = vfmq_laneq_f32(C3, A1, B3, 1);
C3 = vfmq_laneq_f32(C3, A2, B3, 2);
C3 = vfmq_laneq_f32(C3, A3, B3, 3);
vstlq_f32(C+12, C3);
}

Fixed-size 4x4 matrices are chosen because:

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

Summarizing the Neon intrinsics that have been used here:

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

Optimizing a similar case for SVE gives the following code:

```c
void matrix_multiply_nx4_neon(const float32_t *A, const float32_t *B, float32_t *C, uint32_t n) {
    // these are the columns A
    svfloat32_t A0;
    svfloat32_t A1;
    svfloat32_t A2;
    svfloat32_t A3;
    // these are the columns B
    svfloat32_t B0;
    svfloat32_t B1;
    svfloat32_t B2;
    svfloat32_t B3;
    // these are the columns C
    svfloat32_t C0;
    svfloat32_t C1;
    svfloat32_t C2;
    svfloat32_t C3;
    svbool_t pred = svwhilelt_b32_u32(0, n);
    A0 = svld1_f32(pred, A);
    A1 = svld1_f32(pred, A+n);
    A2 = svld1_f32(pred, A+2*n);
    A3 = svld1_f32(pred, A+3*n);
    // Zero accumulators for C values
    C0 = svdup_n_f32(0);
    C1 = svdup_n_f32(0);
    C2 = svdup_n_f32(0);
    C3 = svdup_n_f32(0);
    // Multiply accumulate in 4x1 blocks, that is each column in C
    B0 = svld1rq_f32(svptrue_b32(), B);
    C0 = svmla_lane_f32(C0, A0, B0, 0);
    C0 = svmla_lane_f32(C0, A1, B0, 1);
    C0 = svmla_lane_f32(C0, A2, B0, 2);
    C0 = svmla_lane_f32(C0, A3, B0, 3);
    svst1_f32(pred, C0);
    B1 = svld1rq_f32(svptrue_b32(), B+4);
    C1 = svmla_lane_f32(C1, A0, B1, 0);
    C1 = svmla_lane_f32(C1, A1, B1, 1);
    C1 = svmla_lane_f32(C1, A2, B1, 2);
    C1 = svmla_lane_f32(C1, A3, B1, 3);
    svst1_f32(pred, C1);
    B2 = svld1rq_f32(svptrue_b32(), B+8);
    C2 = svmla_lane_f32(C2, A0, B2, 0);
    C2 = svmla_lane_f32(C2, A1, B2, 1);
    C2 = svmla_lane_f32(C2, A2, B2, 2);
    C2 = svmla_lane_f32(C2, A3, B2, 3);
    svst1_f32(pred, C2);
    B3 = svld1rq_f32(svptrue_b32(), B+12);
    C3 = svmla_lane_f32(C3, A0, B3, 0);
    C3 = svmla_lane_f32(C3, A1, B3, 1);
    C3 = svmla_lane_f32(C3, A2, B3, 2);
    C3 = svmla_lane_f32(C3, A3, B3, 3);
    svst1_f32(pred, C+12, C);
}
```

Summarizing the SVE intrinsics that have been used here:
<table>
<thead>
<tr>
<th>Code element</th>
<th>What is it?</th>
<th>Why are they used</th>
</tr>
</thead>
<tbody>
<tr>
<td>svfloat32_t</td>
<td>An array of 32-bit floats, where the exact number is defined at runtime based on the SVE vector length.</td>
<td>svfloat32_t enables you to use SVE vectors and predicates directly, without relying on the compiler for autovectorization.</td>
</tr>
<tr>
<td>svwhilelt_b32_u32(...)</td>
<td>A function which computes a predicate from two uint32_t integers.</td>
<td>When loading from A and storing to C, svwhilelt_b32_u32(...) ensures you do not read or write past the end of each column.</td>
</tr>
<tr>
<td>svld1_f32(...)</td>
<td>A function which loads 32-bit svfloat32_t floats into an SVE vector.</td>
<td>To get the matrix values needed from A. This also takes a predicate to make sure we do not load off the end of the matrix (unpredicated elements are set to zero).</td>
</tr>
<tr>
<td>svtrue_b32(...)</td>
<td>A function which sets a predicate for 32-bit values to all-true.</td>
<td>When loading from B, svtrue_b32(...) ensures the vector fills completely because the precondition of calling this function is that the matrix has a dimension which is a multiple of four.</td>
</tr>
<tr>
<td>svld1rq_f32(...)</td>
<td>A function which loads an SVE vector with copies of the same 128-bits (four 32-bit values).</td>
<td>To get the matrix values needed from B. Only loads four replicated values because the svmla_lane_f32 instruction only indexes in 128-bit segments.</td>
</tr>
<tr>
<td>svmla_lane_f32(...)</td>
<td>A function which uses the fused multiply accumulate instruction. The function multiplies each 128-bit segment of an svfloat32_t value by the corresponding single element of each 128-bit segment of another svfloat32_t. The svmla_lane_f32(...) function then adds the result to a third svfloat32_t before returning the result.</td>
<td>This operation naturally fits the row-on-column dot products because they are a set of multiplications and additions.</td>
</tr>
<tr>
<td>svst1_f32(...)</td>
<td>A function which stores svfloat32_t at a given address.</td>
<td>To store the results after they are calculated. The predicate ensures we do not store results past the end of each column.</td>
</tr>
</tbody>
</table>

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

**Example 2: Large matrix multiplication with intrinsics**

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

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

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

// these are the columns of a 4x4 sub matrix of A
float32x4_t A0;
float32x4_t A1;
float32x4_t A2;
float32x4_t A3;
// these are the columns of a 4x4 sub matrix of B
float32x4_t B0;
float32x4_t B1;
float32x4_t B2;
float32x4_t B3;
// these are the columns of a 4x4 sub matrix of C
float32x4_t C0;
float32x4_t C1;
float32x4_t C2;
float32x4_t C3;

for (int i_idx=0; i_idx<n; i_idx+=4) {
  for (int j_idx=0; j_idx<m; j_idx+=4) {
    // zero accumulators before matrix op
    C0 = vmovq_n_f32(0);
    C1 = vmovq_n_f32(0);
    C2 = vmovq_n_f32(0);
    C3 = vmovq_n_f32(0);
    // compute base index to 4x4 block
    a_idx = i_idx + n*k_idx;
    b_idx = k*j_idx + k_idx;
    // load most current a values in row
    A0 = vld1q_f32(A+a_idx);
    A1 = vld1q_f32(A+a_idx+n);
    A2 = vld1q_f32(A+a_idx+2*n);
    A3 = vld1q_f32(A+a_idx+3*n);
    // multiply accumulate 4x1 blocks, that is each column C
    B0 = vld1q_f32(B+b_idx);
    C0 = vfmaq_laneq_f32(C0,A0,B0,0);
    C0 = vfmaq_laneq_f32(C0,A1,B0,1);
    C0 = vfmaq_laneq_f32(C0,A2,B0,2);
    C0 = vfmaq_laneq_f32(C0,A3,B0,3);
    B1 = vld1q_f32(B+b_idx+k);
    C1 = vfmaq_laneq_f32(C1,A0,B1,0);
    C1 = vfmaq_laneq_f32(C1,A1,B1,1);
    C1 = vfmaq_laneq_f32(C1,A2,B1,2);
    C1 = vfmaq_laneq_f32(C1,A3,B1,3);
    B2 = vld1q_f32(B+b_idx+2*k);
    C2 = vfmaq_laneq_f32(C2,A0,B2,0);
    C2 = vfmaq_laneq_f32(C2,A1,B2,1);
    C2 = vfmaq_laneq_f32(C2,A2,B2,2);
    C2 = vfmaq_laneq_f32(C2,A3,B2,3);
    B3 = vld1q_f32(B+b_idx+3*k);
    C3 = vfmaq_laneq_f32(C3,A0,B3,0);
    C3 = vfmaq_laneq_f32(C3,A1,B3,1);
    C3 = vfmaq_laneq_f32(C3,A2,B3,2);
    C3 = vfmaq_laneq_f32(C3,A3,B3,3);
    
    // compute base index for stores
    c_idx = n*j_idx + i_idx;
    vstlq_f32(C+c_idx, C0);
    vstlq_f32(C+c_idx+n,C1);
    vstlq_f32(C+c_idx+2*n,C2);
    vstlq_f32(C+c_idx+3*n,C3);
  }
}

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

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

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

Optimizing this code for SVE produces the following code:

```c
void matrix_multiply_sve(const float32_t *A, const float32_t *B, float32_t *C, uint32_t n, uint32_t m, uint32_t k) {
  /*
  * Multiply matrices A and B, store the result in C.
  * It is the users responsibility to make sure the matrices are compatible.
  */
```
This code is almost identical to the earlier Neon code except for the differing intrinsics, and in addition, thanks to predication, there is no longer a constraint on the number of rows of A. However, you must ensure that the number of columns of A and C, and both dimensions of B, are multiples of four because the predication used above does not account for this. Adding such further predication is possible but would reduce the clarity of this example.
Comparing it with the C function and Neon functions, the SVE example:
• Uses WHILELT to determine the predicate for doing each iteration of the outer loop. This guarantees you have at least one element to do by the loop condition.
• Increments i_idx by CNTW (the number of 32-bit elements in a vector) to avoid hard-coding the number of elements done in an iteration of the outer loop.

Program conventions: Macros, Types, and Functions

Macros

In order to use the intrinsics, the Advanced SIMD or SVE architecture must be supported by the compiler. Some specific instructions might not be enabled. When the following macros are defined and equal to 1, the corresponding features are available:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Supported macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neon</td>
<td>• <strong>aarch64</strong></td>
</tr>
<tr>
<td></td>
<td>— Selection of architecture-dependent source at compile time.</td>
</tr>
<tr>
<td></td>
<td>— Always 1 for AArch64.</td>
</tr>
<tr>
<td></td>
<td>• _ARM_NEON</td>
</tr>
<tr>
<td></td>
<td>— Advanced SIMD is supported by the compiler.</td>
</tr>
<tr>
<td></td>
<td>— Always 1 for AArch64.</td>
</tr>
<tr>
<td></td>
<td>• _ARM_NEON_FP</td>
</tr>
<tr>
<td></td>
<td>— Neon floating-point operations are supported.</td>
</tr>
<tr>
<td></td>
<td>— Always 1 for AArch64.</td>
</tr>
<tr>
<td></td>
<td>• _ARM_FEATURE_CRYPTO</td>
</tr>
<tr>
<td></td>
<td>— Crypto instructions are available.</td>
</tr>
<tr>
<td></td>
<td>— Cryptographic Neon intrinsics are therefore available.</td>
</tr>
<tr>
<td></td>
<td>• _ARM_FEATURE_FMA</td>
</tr>
<tr>
<td></td>
<td>— The fused multiply-accumulate instructions are available.</td>
</tr>
<tr>
<td></td>
<td>— Neon intrinsics which use these are therefore available.</td>
</tr>
<tr>
<td>SVE</td>
<td>• _ARM_FEATURE_SVE</td>
</tr>
<tr>
<td></td>
<td>— Always 1 if SVE is supported.</td>
</tr>
<tr>
<td></td>
<td>— The SVE instructions are available.</td>
</tr>
<tr>
<td></td>
<td>— SVE intrinsics which use these are therefore available.</td>
</tr>
</tbody>
</table>

This list is not exhaustive and further macros are detailed on the Arm C Language Extensions web page.

Types
There are three major categories of Neon data type available in `arm_neon.h` which follow these patterns:

- **baseW_t**
  - Scalar data types. For example, `int64_t`.
- **baseWxL_t**
  - Vector data types. For example, `int32x2_t`.
- **baseWxLxN_t**
  - Vector array data types. For example, `int16x4x2_t`.

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

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

There is no existing mechanism that maps directly to the concept of an SVE vector or predicate. The ACLE takes the first approach and classifies SVE vectors and predicates as belonging to a new category of type called **sizeless data types**. Sizeless data types are composed of vector types and predicate types and are pre-pended with `sv`, for example `svint64_t`.

- **baseW_t**
  - Scalar data types. SVE adds support for `float16_t`, `float32_t`, and `float64_t`.
- **svbaseW_t**
  - Sizeless scalar data types for single vectors. For example, `svint64_t`.
- **svbaseWxN_t**
  - Sizeless vector data types for two, three, and four vectors. For example, `svint64x2_t`.
- **svbool_t**
  - Sizeless single predicate data type which has enough bits to control an operation on a vector of bytes.

Where:
- **base** refers to the fundamental data type.
- **bool** refers to the `bool` type from `stdbool.h`.
- **W** is the width of the fundamental type.
- **N** is the number of vector data type instances in a vector array type, for example a struct of arrays of scalars.

### Functions

For Neon, similar to the Arm C Language Extensions, the function prototypes from `arm_neon.h` follow a common pattern. At the most general level, this is:

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

For example:
int8x16_t vmulq_s8 (int8x16_t a, int8x16_t b)

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

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

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

Some of the letters and names are overloaded, but in the order above:

ret

The return type of the function.

v

Short for vector and is present on all the intrinsics.

p

Indicates a pairwise operation. ([value] means value might be present).

q

Indicates a saturating operation (except for vqtb[l][x] in AArch64 operations, where the q
indicates 128-bit index and result operands).

r

Indicates a rounding operation.

name

The descriptive name of the basic operation. Often, this is an Advanced SIMD instruction, but it
does not have to be.

u

Indicates signed-to-unsigned saturation.

n

Indicates a narrowing operation.

q

Postfixing the name indicates an operation on 128-bit vectors.

x

Indicates an Advanced SIMD scalar operation in AArch64. It can be one of b, h, s, or d (that is,8, 16, 32, or 64 bits).

_high

In AArch64, used for widening and narrowing operations involving 128-bit operands. For
widening 128-bit operands, high refers to the top 64-bits of the source operand (or operands).
For narrowing, it refers to the top 64-bits of the destination operand.

_n

Indicates a scalar operand that is supplied as an argument.

_lane

Indicates a scalar operand taken from the lane of a vector. _laneq indicates a scalar operand
taken from the lane of an input vector of 128-bit width. (left | right means only left or
right would appear).
type

The primary operand type in short form.

args

The arguments of the function.

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

svbase[_disambiguator][_type0][_type1]...[_predication]

For example, svc1z[_u16]_m says that the full name is svc1z_u16_m and that its overloaded alias is svc1z_m.

Where:

base

The lower-case name of an SVE instruction, with some adjustments.

_disambiguator

Distinguishes between different forms of a function.

_type0|_type1|...

List the types of vectors and predicates, starting with the return type and continuing with the argument types.

_predication

This suffix describes the inactive elements in the result of a predicated operation. It can be one of z (zero predication), m (merge predication), or x (‘Do not care’ predication).

For more information about the individual function parts, see Arm C Language Extensions for SVE specification.

Resources

Neon:

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

SVE:

- Engineering specifications for the SVE intrinsics can be found in the Arm C Language Extensions for SVE specification.
- The Arm HPC tools for SVE web page describes the tools to enable you to work with SVE code on AArch64.
- Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for Armv8-A
2.4 Case Study: Optimizing HPCG for Arm SVE

This topic presents a case study where the HPCG benchmark application is optimized for Arm SVE.

Note

This topic uses extracts of a blog available on the Arm Community website. To read the blog, see the Optimizing HPCG for Arm SVE.

The HPCG benchmark

The HPCG (High Performance Conjugate Gradients) benchmark solves a linear system of equations by using a preconditioned conjugate gradient method. The most interesting bits of this benchmark are the characteristics of the computations that are performed within its kernels, which are representative of real-world scientific applications that are run on High Performance Computing (HPC) systems. The benchmark exercises all aspects of the compute system and emphasizes the significance of both compute and data-delivery subsystem (memory, storage, and interconnect) to the overall performance.

Arm has been working on optimizing HPCG because of the importance of this benchmark in the HPC community. These optimizations targeted the lack of parallelism present in the Gauss-Seidel kernel. Detailed information about the parallelization techniques that are applied can be found in a blog about Parallelizing HPCG.

As a result of the parallelization work, some single-core performance is lost. To recover the single-core performance, this case study looks at how to vectorize the main HPCG kernels, and port them to the Arm Scalable Vector Extension (SVE).

Tooling and Arm® Instruction Emulator

With no public hardware available for the Arm Scalable Vector Extension (SVE), emulation or simulation must be used.

This study uses emulation, and uses the Arm Instruction Emulator (ArmIE) tool, because:

1. ArmIE is publicly available.
2. ArmIE is a fast emulator for SVE operations, allowing larger workloads to be run compared to using a simulator.
3. ArmIE allows extensible application instrumentation through custom plug-ins.
4. There is an absence of tuned SVE performance models in simulators.

The Arm Instruction Emulator (ArmIE) enables you to execute unsupported instructions on Armv8-A platforms, such as those from the SVE instruction set, by dynamically converting those instructions into native ones. However, because of this conversion, the timing information is lost.

In addition to emulation, ArmIE can be expanded using dynamic binary instrumentation clients. These clients can be used to extract different metrics such as dynamic instruction counts or memory and instructions traces. ArmIE supports an emulation API that enables you to write your own clients, therefore expanding the ArmIE instrumentation capabilities further.

ArmIE comes with four SVE-ready instrumentation clients:

- SVE Inscount: Instruction counter
- SVE Opcodes: Opcodes counter
- SVE Instrace: Instruction trace
- SVE Memtrace: Memory trace

You can find further information on how ArmIE works and how to use these clients in Analyze Emulated SVE on Existing Arm®v8-A Hardware on page 5-81.
The Arm SVE methodology

When optimizing HPCG, it is important to infer the potential relative performance benefits from the metrics that are offered by ArmIE using its clients. To achieve these performance benefits, we apply a flexible methodology where the steps can be completed in any order.

To help the optimization process, metrics are obtained for the whole application, and also for specific parts of the code (Regions of Interest (RoI)). The ArmIE memory trace client already supports RoI instrumentation, and more clients will also support RoI instrumentation, in future versions. For this work, the RoI functionality is added to all the clients.

The metrics for this analysis are:

- **Compiler auto-vectorization analysis**
  The ratio of SVE instructions in your code directly tells you if your vector units are used at all. Unless you use SVE intrinsics or hand written assembler, vectorization relies on the compiler. Therefore, it is important to check what the compiler is able to vectorize, because it can point you to problematic areas of the code (for example, loop `<x>` is not vectorizing because it is reversed).

- **Instruction counts**
  The compiler is able to vectorize most of the loops found in the main computational kernels, but it is useful to know how many of these instructions are present in comparison to the total number of instructions executed.

- **Vector lane utilization and memory accesses breakdown**
  Analyzing the average lane utilization of the vectors can help identify performance issues. SVE uses predicate registers to specify which lanes in the vector are enabled or not. Disabled lanes do not update their destination register values. Therefore, even if SVE instructions are issued, if the average number of enabled lanes per vector instruction is low, vectors are not fully utilized and it results in lower performance. The vector utilization metric can be derived from the memory traces the ArmIE memory trace client generates.
The memory instruction mix (how many times each kind of memory access has occurred) can also be derived from the memory traces generated with ArmIE. The memory instruction mix metric provides the ratio of SVE memory instructions compared with non-SVE instructions. In addition, for each of these SVE memory accesses, the kind of memory access is reported (contiguous or gather/scatter access). The type of access is obtained by post-processing the memory traces generated by ArmIE. For all the memory accesses, the number of bytes loaded or stored is also available.

- Cache simulations

Cache statistics can tell you if your code can perform better or not. Cache statistics require a cache model. ArmIE, as an emulator, does not have a cache model. Instead, for this tutorial, a cache simulator which supports prefetchers (that are implemented as plug-ins), was written. The simulator uses a stride prefetcher.

---

**Note**

The cache simulator, scripts, and ArmIE plugins used in this work are available as open source software in the [Arm-software GitHub space](https://github.com/ARM-software).

The methodology that is used to perform the data analysis:
1. Run the applications using ArmIE. ArmIE collects the metric information previously described.
2. Analyze the results.
3. Infer the relative performance variations.

**HPCG versions**

To improve the single-core performance of our optimized HPCG code, a version of HPCG with SVE intrinsics was also developed (in addition to the version with optimized code).

This extra version of HPCG allows the comparison of three versions (all compiled with an SVE-capable compiler). For readability, the following naming scheme is used:
1. Baseline (reference HPCG code)
2. No-intrinsics (optimized HPCG code without SVE intrinsics. For more information about this code, see [Parallelizing HPCG](#)).
3. Intrinsics (optimized HPCG code with SVE intrinsics)

**Compiler auto-vectorization analysis**

To compare compiler vectorization, HPCG is compiled with different compilers and the number of automatically vectorized loops is noted. To understand the differences with other SIMD technologies, an AVX2-enabled compiler is also used for comparison.

<table>
<thead>
<tr>
<th>Compiler</th>
<th># Vectorized loops</th>
<th>baseline</th>
<th>no-intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE</td>
<td>GCC 8.2.0</td>
<td>8/8</td>
<td>12/17</td>
</tr>
<tr>
<td></td>
<td>Arm HPC Compiler 19.0</td>
<td>8/8</td>
<td>12/17</td>
</tr>
<tr>
<td>AVX2</td>
<td>Intel Compiler 19.0.3</td>
<td>8/8</td>
<td>17/17</td>
</tr>
</tbody>
</table>

**Figure 2-4 Compiler autovectorization comparison**

**Result**

The compiler left some loops that are unvectorized in the no-intrinsics code. Those unvectorized loops are contained in the most executed kernel, the symmetric Gauss-Seidel (in other words, SymGS). The identification of these loops helps prioritize which loops to hand-optimize first.
Instruction counts

To gather the instruction count information, the instruction count client (shipped with ArmIE) is used, and the RoI is restricted to one conjugate gradient iteration.

Result

The following chart shows the breakdown of the dynamically executed instructions for each version, differentiating between the SVE and the non-SVE instructions.

![Figure 2-5 Instruction count reduction when increasing vector length](image)

The optimized versions of HPCG executed more instructions than the baseline code. This is expected because of the overhead that is caused by the parallelization techniques applied. With handcrafted SVE intrinsics, you can reduce the total number of dynamically executed instructions (compared to the HPCG version without intrinsics), reducing the gap compared with the reference code.

Looking at the percentage of SVE instructions against non-SVE instructions, the intrinsics code presents a lower ratio than the other two versions. This is unexpected. To understand more, we gathered the instruction counts with ArmIE at the kernel level:
From the graph above, you can see that:

- Vectorization is more evenly present across the kernels in the intrinsics version.
- The multi-grid kernel presents a lower percentage of SVE instructions executed.
- The dot-product vector instruction ratio is also lower.
- SPMV and WAXPBY present a similar ratio compared to the other two versions of HPCG.

Looking at the multi-grid kernel case, the reason for the lower ratio is explained by a more efficient use of SVE instructions. The total number of instructions is reduced compared to the no-intrinsics code. The DotProduct kernel presents a similar behavior as the multi-grid, the intrinsics code features a lower percentage of SVE instructions, but at the same time, the number of total instructions that are dynamically executed is also lower.

As for the WAXPBY, the compiler generates both SVE and non-SVE versions of the code. The version of the kernel that is executed is decided at runtime. In all the executions that are performed, the non-vectorized version of the kernel is always chosen.

**Vector lane utilization and Memory accesses breakdown**

**Vector lane utilization**

After understanding how much vectorization is present in the code, it is important to find the vector utilization. To get this information, run the three versions of HPCG through the memory trace client in ArmIE. To obtain the number of lanes that are enabled for each SVE memory access, post-process the generated memory traces.

**Result**
All three versions of the benchmark present the same characteristics:

- Around 10% of the SVE memory accesses have 0% to 33% of their lanes enabled.
- Around 15% of the SVE memory accesses have 34% to 99% active lanes.
- Around 75% of the SVE memory accesses are instructions where all the lanes were enabled.

Assuming a similar vector utilization for non-memory SVE operations, you can infer that the vectors are fully utilized most of the time, averaging a vector lane utilization of ~82% for all HPCG versions.

**Memory accesses breakdown**

Although the SVE memory accesses present a good average vector lane utilization, you cannot expect the same latency for all kinds of SVE memory accesses, in other words, contiguous, and gather-load or scatter-store accesses. In general, a good approach to increase performance is to try to avoid the use of gather-loads or scatter-stores because they can potentially access a higher number of different cache lines, and therefore are more resource demanding. To collect the gather-load and scatter-store information, we must perform further post-processing analysis on the memory traces, and count the number of different memory accesses.

**Result**
The memory instruction breakdown is similar for all three versions of the code, with the intrinsics code presenting a higher ratio of SVE memory accesses and a lower percentage of non-SVE memory accesses, when compared to the other two versions.

For the different memory accesses present in the code, they are split between:
- SVE contiguous memory accesses.
- SVE gather-loads and SVE scatter-stores memory accesses.
- Non-SVE memory accesses.

The chart also distinguishes between SVE memory accesses with all lanes active, or some of the lanes disabled. Around 60% of the memory accesses are generated by SVE memory instructions, with half of those being contiguous accesses with all lanes enabled. SVE gather-load and scatter-store accesses represent around 20% of the all memory accesses.

**Cache simulations**

To complement the memory tracing analysis, we run the traces on the cache simulator.

In these experiments, the simulator is configured with these parameters:
When implementing the parallelization techniques to optimize the reference HPCG code, there is a potential cache hit ratio degradation. This behavior with the cache simulator is observed and reflected in the chart below. Interestingly, the hit ratio is improved in the intrinsics version, when compared to the optimized HPCG code without intrinsics.

The increase in L1 hit ratio also translates into a lower average number of cycles per memory access, compared with the optimized HPCG code without SVE intrinsics:

Putting everything together

So far in the analysis, each metric has been presented separately. All of the metrics are obtained with different ArmIE clients and different post-processing procedures. Although all these metrics present
value on their own, they should not be used independently to assess potential performance variations. Instead, you should look at all the metrics combined.

In these examples, three different versions of HPCG are compared. Looking at the metrics that are obtained, the observed changes in both optimized codes (with and without intrinsics) compared to the reference HPCG implementation, are summarized:

<table>
<thead>
<tr>
<th></th>
<th>Optimized HPCG</th>
<th>With intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE instruction ratio</td>
<td>Similar to reference code.</td>
<td>SVE instructions are present in all computational kernels.</td>
</tr>
<tr>
<td>Vector lane utilization</td>
<td>Same as in the reference code.</td>
<td>Same as in the reference code.</td>
</tr>
<tr>
<td>Memory instructions</td>
<td>Same as in the reference code.</td>
<td>More SVE memory accesses and less non-SVE ones are executed.</td>
</tr>
<tr>
<td>Cache hit-ratio</td>
<td>Lower than in the reference code.</td>
<td>Improved hit-ratio compared to no-intrinsics code.</td>
</tr>
</tbody>
</table>

Figure 2-11 Optimization summary

From the results that are obtained, a loss of single-core performance is expected in the optimized HPCG code without intrinsics, versus the reference code. This can be seen by the vector lane utilization and the types of memory instructions, with their values being similar to the reference code, as well as the higher number of instructions that are executed, and higher cache miss ratio.

For the intrinsics code, you can infer a potential performance gain compared to the optimized HPCG without SVE intrinsics, because of the:

- Similar average vector lane utilization.
- Lower number of total instructions.
- Higher number of SVE memory accesses.
- Lower number of non-SVE memory accesses.
- An improved cache hit ratio.

When comparing against the reference code, it is unclear which presents a higher performance. The reference code executes fewer instructions and the cache hit ratio is higher, whereas the intrinsics code presents a better memory instruction mix and higher ratio of SVE instructions per computational kernel. Because HPCG is known to be heavily memory bound, you would expect a better performance with the intrinsics version because the memory instructions breakdown presents more favorable characteristics.

Conclusions

HPCG is used as an example to illustrate the methodology you can use to optimize applications for SVE in the absence of tuned performance models or real hardware. The methodology relies on ArmIE and its clients, which in this work, are extended to provide metrics necessary for the analysis. ArmIE development continues, and you can expect new and more refined clients, as well as more features and stability in future versions.

Related information

Arm-software GitHub space
Parallelizing HPCG
2.5 Additional resources

This topic lists the resources that describe the Arm Scalable Vector Extension (SVE) in more detail.

Want to evaluate SVE?

To compile and run SVE code on non-SVE platforms, download and install Arm Compiler for Linux and Arm Instruction Emulator.

SVE resources:

- Past presentations and Hackathon materials
  Past presentations at Arm events, including downloadable SVE Hackathon materials.

- 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.

- White Paper: Arm Scalable Vector Extension and application to Machine Learning
  In this white paper, code examples are presented that show how to vectorize some of the 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 Extensions (ACLE) 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 some SVE programming tips that are supported by assembly examples that use the Arm C Language Extensions (ACLE) for SVE.

It contains the following sections:

- **3.1 Vector Length Agnostic (VLA) programming** on page 3-46.
- **3.2 For and While loop vectorization** on page 3-51.
- **3.3 Do-while loop SVE vectorization** on page 3-54.
- **3.4 Effective vector length bandwidth utilization tips** on page 3-56.
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. Only specifying a maximum size enables different Arm architectural licensees to develop their own implementation, and target specific workloads and technologies which could benefit from a particular vector length.

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

The SVE features require a new programming style, called Vector Length Agnostic (VLA) programming. The following examples describe this programming approach in practice.

Example

This example compares a code snippet that uses simple C loop processing integers, example01, with Neon (example01_neon), and SVE (example01_sve), variants of the same code.

To vectorize a loop (see the example01 function in example01) for a traditional SIMD architecture, you (or the compiler) need to know how many elements the vector loop can process in one iteration.

example01:

```c
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];
}
```

Using the Arm Neon (C intrinsics), you can create a vectorized version of example01 (see example01_neon).

example01_neon (Neon code for example01):

```c
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];
}
```

In example01_neon the loop operates on four elements, in other words, as many 32-bit ints as a Neon vector register can hold. Furthermore, where for other traditional unpredicated SIMD architectures, the programmer (or the compiler) must add an extra loop (called a loop tail) that is responsible for processing those iterations at the end of the loop that do not fit in a full vector length.

With SVE, the ‘fixed-width’ approach is not appropriate. example01_sve shows an assembly version of example01, with SVE instructions:

example01_sve (VLA SVE code for example01):

```assembly
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
mov x4, 0 # set 'i=0'
cond # branch to 'cond'
loop_body:
ldlw z0.s, p0/z, [x1, x4, lsl 2] # load vector z0 from address 'b + i'
ldlw 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
stlw 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:
```

101726_0200_00_en Copyright © 2019 Arm Limited or its affiliates. All rights reserved. Non-Confidential
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.

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` 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]
```

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 if the first (LSB) lane of `p0.s` is set to true, that is, if 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, and `b.any` checks 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 instructions to test the condition flags set by SVE instructions, are:

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>SVE interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.none</td>
<td>No active elements are true.</td>
</tr>
<tr>
<td>b.any</td>
<td>An active element is true.</td>
</tr>
<tr>
<td>b.nlast</td>
<td>The last active element is not true.</td>
</tr>
<tr>
<td>b.last</td>
<td>The last active element is true.</td>
</tr>
<tr>
<td>b.first</td>
<td>The first active element is true.</td>
</tr>
<tr>
<td>b.nfirst</td>
<td>The first active element is not true.</td>
</tr>
<tr>
<td>b.pmore</td>
<td>An active element is true but not the last element.</td>
</tr>
<tr>
<td>b.plast</td>
<td>The last active element is true or none are true.</td>
</tr>
<tr>
<td>b.tcont</td>
<td>Scalarized CTERM loop termination not detected.</td>
</tr>
<tr>
<td>b.tstop</td>
<td>Scalarized CTERM loop termination detected.</td>
</tr>
</tbody>
</table>

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

`example01_sve_pseudo_c` (C pseudocode for the blocks in `example01_sve`):

```c
while( /* cond */ ) {
    /* loop */
}
```
The loop body is also VLA. In each iteration, the operations that are 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 four, 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 several 32-bit words in the machine-implemented vector length with incw.

All the instructions in the loop_body are predicated with p0 (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 that is used in the code. This loop executes correctly on any implementation of SVE, because incw increases the loop iterator according to the current SVE vector length.

To compare, example01_neonassembly shows an assembly version of the Neon code in example01_neon, and shows both the vector body and the loop tail blocks.

**example01_neonassembly** (Neon assembly code that is generated from the C intrinsics in example01_neon):

```assembly
# 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
  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

.b.lo .vector_body:
  cmp x8, x3               # compare the loop counter with N
  b.hs .function_exit      # if greater or equal N, terminate

.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]  # load 2 elements from 'b+i'
  ldr w13, [x2, x8, lsl 2]  # load 2 elements from 'c+i'
  add w12, w13, w12         # add the vector
  str w12, [x0, x8, lsl 2]  # store 2 elements in 'a+i'
  cmp x8, x3               # 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.

**example02** (a loop with conditional execution):

```c
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];
}
```

**example02_sve** (SVE vector version of example02):

```assembly
# 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
    # pl.s[0] = z4.s[0] > 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    stlw    z0.s, p1, [x0, x5, lsl 2]
incw    x5
cond:
    whilelt p0.s, x5, x3
b.ne    loop_body
ret

Note

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

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 that is 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 does not change the inactive lanes.

The examples in example03_c and example03_sve show how merging predication can be used to perform a conditional reduction.

eample03_c (a reduction):

```
int example03(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;
}
```

eample03_sve (SVE vector version of example03_c):

```
  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 that is introduced with SVE is the gather load / scatter store set of instructions, which allows it to operate on non-contiguous data in memory. example04 shows an example of this non-contiguous data in memory.
**example04** (loads data from an array of addresses):

```c
void example04(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 **example04_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 that are stored in `z1.s` are interpreted as 32-bit scaled indices, and sign extended (sxtw) to 64-bit before being left-shifted by two 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.

**example04_sve** (SVE vectorized version of **example04**):

```assembly
mov     x5, 0
b       cond
loop:
    ld1w    z1.s, p0/z, [x4, x5, lsl 2] # load a vector
    ld1w    z0.s, p0/z, [x1, z1.s, sxtw 2] # 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:

```c
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:

```c
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 the code examples that are presented above calculate the same addition of two arrays of integers.

For/While loop Neon™ vectorization

In the following example code, 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 four, you must implement the scalar loop (the loop at lines 16-22) to calculate the remaining results, if there are any.

```
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
```

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.

Like the Advanced SIMD vectorization, the processing lane width is 32-bits. On line four, the governing predicate \( p1 \) initializes for 32-bit elements with the instruction \( \text{WHILELT} \). The \( \text{WHILELT} \) 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 \( \text{INCW} \), this value is incremented by the number of 32-bit elements in the implemented vector length (line 12).

Note

The number of output results that are 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 \( \text{WHILELT} \) 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 \( \text{B.FIRST} \) (line 14), manages whether there is another input array element to process.

The SVE vector length agnostic vectorization code can be rewritten in C with the Arm C language extension (ACLE) for SVE as:

```c
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(svtrue_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, [x8, 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 example, prefetch instructions with PLDL1STRM specifier (lines 2-7, 20-21) signals 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, so it does not 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 extra 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

This topic is an example of a simple vectorization of the *Do-while* loop with the SVE instruction set using a Vector Length Agnostic (VLA) approach.

The *Do-while* loop that computes the sum of the first \( N \) squares:

```c
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.

```asm
    PTRUE p1.s
    MOV z5.s, #0
    MOVI d6, #0

    INDEX z0.s, x0, #-1
    CMPGT p0.s, p1/Z, z0.s, #0
    B.NONE .L_result

    .L_loopStart:
    MLA z5.s, p0/M, z0.s, z0.s
    DECW z0.s
    CMPGT p0.s, p1/Z, z0.s, #0
    B.FIRST .L_loopStart

    UADDV d6, p1, z5.s

    .L_result:
    STR s6, [x1]
    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 SVE vector length agnostic vectorization code can be rewritten in C with the *Arm C language extension (ACLE) for SVE* as:

```c
void example_sum_squares( int N, int * sum) {
    svbool_t pred_N;
    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 = svcmpeq( 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 = svcmpeq( p_all, svN_tmp, 0);
    } while ( svptest_first( p_all, pred_N));
}
*sum = (int) svadd( 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:

```c
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:

```
1       MOV x5, #0
2       SXTW x0, w0
3       WHILELT p4.s, x5, x0
4       B.NONE .L_return
5
6       SXTW x1, w1
7       ADD x1, x3, x1, LSL #1
8       PTRUE p5.s
9
10  .L_OuterLoop:
11      MOV x6, #0
12      MOV x7, x3
13      LDISH z10.s, p4/Z, [x2, x6, LSL #1]
14      LD1RSH z1.s, p5/Z, [x7]
15      ADD x6, x6, #1
16      ADD x7, x7, #2
17      MUL z10.s, p4/M, z10.s, z1.s
18      CMP x7, x1
19      B.EQ .L_InnerLoopEnd
20
21  .L_InnerLoop:
22      LDISH z2.s, p4/Z, [x2, x6, LSL #1]
23      LD1RSH z1.s, p5/Z, [x7]
24      ADD x6, x6, #1
25      ADD x7, x7, #2
26      MLA z10.s, p4/M, z2.s, z1.s
27      CMP x7, x1
28      B.MI .L_InnerLoop
29
30  .L_InnerLoopEnd:
31      ASR z10.s, p4/M, z10.s, #16
32      ST1H z10.s, p4, [x4]
33      INCH x4
34      INCH x2
35      INCW x5
36      WHILELT p4.s, x5, x0
37      B.FIRST .L_OuterLoop
38```
3 SVE Vector Length Agnostic programming
3.4 Effective vector length bandwidth utilization tips

39 .L.return:
40      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 on page 4-59.
• 4.2 Vector Dot-Product on page 4-61.
• 4.3 FIR Filter on page 4-63.
• 4.4 Matrix Multiplication on page 4-65.
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 4-59.

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:

```c
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:

```assembly
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      PTUE 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.

Next, 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, the first
occurrence of the maximum vector element is determined. With the SEL instruction at line 27, all indexes of the maximum vector element are kept in the vector register z10. The indexes corresponding to other non-maximum vector elements are overwritten by the value 0xFFFF. Finally, the first occurrence of the maximum vector element is obtained by performing the unsigned minimum reduction across indexes that are 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 adequately replacing 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 4-59
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 4-61.
- 4.2.2 Vectors Dot-Product with Complex SP Floating-Point Elements and Result on page 4-61.

4.2.1 Vector Dot-Product Calculation

This topic illustrates the vector dot-product equation.

Vector dot-product is computed as:

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

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

This example is the optimized SVE implementation of dot-product computation of two vector elements with complex SP floating-point input data and result.

```plaintext
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      WHILELET p4.b, aPtr, size
29      B.FIRST .L_unrolled_loop_vecdot
30
31  .L_tail_vecdot:
32      DECB aPtr
33      WHILELET 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
```
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) and B.FIRST (line 29) check if there are more than one vector length input elements remaining, and only where the vectorized loop start (line 15) is entered. Otherwise, the termination code processing the remaining input elements of the first vector length group is executed (lines 31-40).

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

\[ \alpha = \alpha_r + i * \alpha_i, \beta = \beta_r + i * \beta_i \]

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:

\[ c_r+ = -\alpha_i * \beta_r, c_i+ = \alpha_i * \beta_r \]

Therefore:

\[ c+ = (\alpha_r * \beta_r - \alpha_i * \beta_i) + i * (\alpha_r * \beta_i + \alpha_i * \beta_r) \]

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 4-61
4.2.2 Vectors Dot-Product with Complex SP Floating-Point Elements and Result on page 4-61
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 4-63.
- **4.3.2 FIR filtering with real SP floating-point elements** on page 4-63.

### 4.3.1 FIR Filtering Theory and C Implementation

This topic illustrates the implementable FIR filtering theory.

The implementable FIR filtering is represented by:

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

The C code that implements the FIR filtering is:

```c
#include <cstdint>

#define FLOAT

float32_t acc;

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

### 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 SP floating-point multiply FMUL instruction (line 20), and the SP floating-point multiply-accumulate FMLA instruction (line 29), implement the FIR filtering computation.

```assembly
1 MOV x6, #0 // int32_t n
2 MOV x8, hPtr // float32_t * h
3 ADD size, yPtr, size, LSL #2
4 SUB hPtr, x1 // float32_t * x
5 SUB yPtr, x2 // float32_t * y
6 MOV x3, #0 // int32_t t
7 ADD x3, #t
8 SUB hPtr, x2 // float32_t * x
9 SUB yPtr, x1 // float32_t * y
10 MOV x4, #0 // int32_t t
11 SUB hPtr, x1 // float32_t * x
12 SUB yPtr, x2 // float32_t * y
13 L_FIR_outer_loop:
14 MOV x6, #0
15 MOV x8, hPtr
```
16     LD1W z2.s, p4/z, [xPtr, x6, LSL #2]
17     LD1RW z1.s, p5/z, [x8]
18     ADD x6, x6, #1
19     ADD x8, x8, #4
20     FMUL z10.s, z2.s, z1.s
21
22  .L_FIR_inner_loop:
23     LD1W z2.s, p4/z, [xPtr, x6, LSL #2]
24     // contiguous load of input data
25     LD1RW z1.s, p5/z, [x8]
26     // load with broadcast of one FIR filter coefficient
27     ADD x6, x6, #1
28     ADD x8, x8, #4
29     FMLA z10.s, p5/m, z2.s, z1.s
30     CMP x8, taps
31     B.MI .L_FIR_inner_loop
32
33     ST1W z10.s, p4, [yPtr]
34     INCB yPtr
35     ADDVL xPtr, xPtr, #1
36     WHILELT p4.b, yPtr, size
37     B.FIRST .L_FIR_outer_loop
38
39  .L_return:
40     RET

Related concepts
4.3.1 FIR Filtering Theory and C Implementation on page 4-63
4.3.2 FIR filtering with real SP floating-point elements on page 4-63
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 4-65.
- 4.4.2 Matrix Multiplication with Real HP Floating-Point Elements on page 4-66.
- 4.4.3 Matrix Multiplication with Real Fixed-Point 8-Bit Input Elements and Real Fixed-Point 32-Bit Output Elements on page 4-70.

4.4.1 Matrix Multiplication with Real DP Floating-Point Elements

This topic illustrates a Double Precision (DP) floating-point matrix multiplication equation, and shows example code that implements the equation.

The DP floating-point matrix multiplication:

\[
\text{out}[M, N] = \text{inLeft}[M, K] \ast \text{inRight}[K, N]
\]

The following C code shows the DP floating-point matrix multiplication equation implemented with no previous rearrangement of input matrices:

```c
#include <math.h>

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 that are 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. This example is a vector length agnostic implementation, for vector lengths that are powers of 2, and supported by up to 1024-bits:

```c
#include <math.h>
#include <arm_sve_sve2.h>

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;

for (x=0; x<M; x+=4) {
    ptrOUT = &out[x*N];

    for (y=0; y<N; y+=vl) {
        acc0 = svdup_f64(0.0);
        acc1 = svdup_f64(0.0);
        acc2 = svdup_f64(0.0);
        acc3 = svdup_f64(0.0);

        ptrIN_left = &inLeft[x*K];
        ptrIN_right = &inRight[y];

        for (z=0; z<K; z+=2) {
            inR_0 = svld1(p64_all, ptrIN_right);
            inR_1 = svld1(p64_all, &ptrIN_right[offsetOUT_1]);

            inL_0 = svld1rq(p64_all, ptrIN_left);
            inL_1 = svld1rq(p64_all, &ptrIN_left[offsetIN_1]);
            inL_2 = svld1rq(p64_all, &ptrIN_left[offsetIN_2]);
            inL_3 = svld1rq(p64_all, &ptrIN_left[offsetIN_3]);

            acc0 = svmla_lane(acc0, inR_0, inL_0, 0);
            acc0 = svmla_lane(acc0, inR_1, inL_0, 1);

            acc1 = svmla_lane(acc1, inR_0, inL_1, 0);
            acc1 = svmla_lane(acc1, inR_1, inL_1, 1);

            acc2 = svmla_lane(acc2, inR_0, inL_2, 0);
            acc2 = svmla_lane(acc2, inR_1, inL_2, 1);

            acc3 = svmla_lane(acc3, inR_0, inL_3, 0);
            acc3 = svmla_lane(acc3, inR_1, inL_3, 1);

            ptrIN_right += 2*N;
            ptrIN_left += 2;
        }

        svst1(p64_all, ptrOUT, acc0);
        svst1(p64_all, &ptrOUT[offsetOUT_1], acc1);
        svst1(p64_all, &ptrOUT[offsetOUT_2], acc2);
        svst1(p64_all, &ptrOUT[offsetOUT_3], acc3);

        ptrOUT += vl;
    }
}

The SVE vectorization is implemented by unrolling:

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

The innermost loop, at line 37, produces results of a sub-block:

\[
\text{out}[4, vl] = \text{inLeft}[4, K] * \text{inRight}[K, vl]
\]

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:

\[
\text{out}[M, N] = \text{inLeft}[M, K] * \text{inRight}[K, N]
\]

This HP floating-point matrix multiplication equation is implemented in two steps:
1. The left-matrix is rearranged and tailored to the data processing of the second step.

2. The dot-product calculations are performed to produce the final matrix multiplication results.

In the first step, the entire left-matrix is rearranged; each block of 8 full rows is transposed. The following C code shows this rearrangement:

```c
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.

The following C code shows these calculations:

```c
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 the matrices being 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:

```c
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();

  uint64_t x, y, z;
  float16_t *ptr_in;
  float16_t *ptr_out;
  for (x=0; x<M; x++) {
    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++;
      }
    }
  }
}
```
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, where vector lengths are 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:

```c
void matmul_dotp_fp16( uint64_t M, uint64_t K, uint64_t N,
                        float16_t * inLeft_MOD, float16_t * inRight, float16_t * out) {
    uint64_t x, y, z;
    svfloat16_t inR, inL;
    svfloat16_t acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
    svbool_t p ld_st;
    uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3, offsetOUT_4;
    uint64_t offsetOUT_5, offsetOUT_6, offsetOUT_7;
    svbool_t p16_all = svpttrue_b16();
    uint64_t vl = svcnth();

    float16_t *ptrIN_left;
    float16_t *ptrIN_right;
    float16_t *ptrOUT;

    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];
        for(y=0; svptest_first( p16_all, p ld_st = svwhilelt_b16(y, N)); y+=vl)
        {
            ptrIN_left = &inLeft_MOD[x*K];
            ptrIN_right = &inRight[y];

            acc0 = svdup_f16(0.0);
            acc1 = svdup_f16(0.0);
            acc2 = svdup_f16(0.0);
            acc3 = svdup_f16(0.0);
            acc4 = svdup_f16(0.0);
            acc5 = svdup_f16(0.0);
            acc6 = svdup_f16(0.0);
            acc7 = svdup_f16(0.0);

            for (z=0; z<K; z++) {
                inR = svld1(p ld_st, &ptrIN_right[z*N]);
                inL = svld1rq(p16_all, &ptrIN_left[8*z]);

                acc0 = svmlaLane(acc0, inR, inL, 0);
                acc1 = svmlaLane(acc1, inR, inL, 1);
                acc2 = svmlaLane(acc2, inR, inL, 2);
                acc3 = svmlaLane(acc3, inR, inL, 3);
                acc4 = svmlaLane(acc4, inR, inL, 4);
                acc5 = svmlaLane(acc5, inR, inL, 5);
                acc6 = svmlaLane(acc6, inR, inL, 6);
                acc7 = svmlaLane(acc7, inR, inL, 7);
            }

            svst1(p ld_st, ptrOUT, acc0);
            svst1(p ld_st, &ptrOUT[offsetOUT_1], acc1);
            svst1(p ld_st, &ptrOUT[offsetOUT_2], acc2);
            svst1(p ld_st, &ptrOUT[offsetOUT_3], acc3);
            svst1(p ld_st, &ptrOUT[offsetOUT_4], acc4);
            svst1(p ld_st, &ptrOUT[offsetOUT_5], acc5);
            svst1(p ld_st, &ptrOUT[offsetOUT_6], acc6);
            svst1(p ld_st, &ptrOUT[offsetOUT_7], acc7);
        }
        ptrOUT += vl;
    }
}
```

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:

\[ out[8, vl] = inLeft[8, K] \ast inRight[K, vl] \]
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, v1], results are stored to the memory by contiguous vector length store ST1H (lines 56-63).

With the matrix dimensions 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_1d_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

This topic illustrates a fixed-point matrix multiplication equation, with real 8-bit input elements and real 32-bit output elements, and shows example code that implements the equation.

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

\[ \text{out}[M, N] = \text{inLeft}[M, K] \times \text{inRight}[K, N] \]

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.

The following C code shows these rearrangements:

```c
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 sub-blocks of size \([4, v1/4]\), where v1 is the number of 8-bit elements in a machine vector length. Next, the following C code shows that each sub-block is transposed, and stored:

```c
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 (x=0; x<K; x+=4) {
        ptr_out = &inRight_MOD[y*K];
        for (y=0; y<N; y+=vl_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 following C code shows the calculations:

```c
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 that are 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.

This example code is an SVE implementation of the left-matrix rearrangement:

```c
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++) {
        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]);

t8 = svzip1(svreinterpret_u32(r0), svreinterpret_u32(r4));
t9 = svzip1(svreinterpret_u32(r2), svreinterpret_u32(r6));
t10 = svzip1(svreinterpret_u32(r1), svreinterpret_u32(r5));
t11 = svzip1(svreinterpret_u32(r3), svreinterpret_u32(r7));
t12 = svzip2(svreinterpret_u32(r0), svreinterpret_u32(r4));
t13 = svzip2(svreinterpret_u32(r2), svreinterpret_u32(r6));
t14 = svzip2(svreinterpret_u32(r1), svreinterpret_u32(r5));
t15 = svzip2(svreinterpret_u32(r3), svreinterpret_u32(r7));
t16 = svzip1(t8, t9);
t17 = svzip1(t10, t11);
t18 = svzip2(t8, t9);
t19 = svzip2(t10, t11);
r00 = svzip1(t16, t17);
r11 = svzip1(t18, t19);
r22 = svzip2(t16, t17);
r33 = svzip2(t18, t19);
t16 = svzip1(t12, t13);
t17 = svzip1(t14, t15);
t18 = svzip2(t12, t13);
t19 = svzip2(t14, t15);
r44 = svzip1(t16, t17);
r55 = svzip2(t16, t17);
r66 = svzip1(t18, t19);
r77 = svzip2(t18, t19);

    switch(nb_st elems) {
        default :
            svst1_vnum(p8_all, ptrOUT, 7, svreinterpret_u8(r77));
            svst1_vnum(p8_all, ptrOUT, 6, svreinterpret_u8(r66));
            svst1_vnum(p8_all, ptrOUT, 5, svreinterpret_u8(r55));
            svst1_vnum(p8_all, ptrOUT, 4, svreinterpret_u8(r44));
            svst1_vnum(p8_all, ptrOUT, 3, svreinterpret_u8(r33));
            svst1_vnum(p8_all, ptrOUT, 2, svreinterpret_u8(r22));
            svst1_vnum(p8_all, ptrOUT, 1, svreinterpret_u8(r11));
            svst1(p8_all, ptrOUT, svreinterpret_u8(r00));
        case 4 :
    }
    ptrIN += vl;
    ptrOUT += 8*vl;
    nb_st elems -= 8;
} }

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).

With the matrix dimensions a multiple of 32, and the element size being 8-bit, where vector lengths are 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).

This example code is an SVE implementation of right-matrix rearrangement:
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;
offsetOUT_3 = K*vl_4*3;

nb_st_elems = 4*N/vl;

for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, N)); y+=vl) {
    ptrOUT = &inRight_MOD[y*K];
    for (x=0; x<K; x+=4) {
        ptrIN = &inRight[x*N+y];
        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]);
        svuint8_t t4 = svzip1(r0, r2);
        svuint8_t t5 = svzip1(r1, r3);
        svuint8_t t6 = svzip2(r0, r2);
        svuint8_t t7 = svzip2(r1, r3);
        r0 = svzip1(t4, t5);
        r1 = svzip2(t4, t5);
        r2 = svzip1(t6, t7);
        r3 = svzip2(t6, t7);
        switch(nb_st_elems) {
        default :
            svst1(p8_all, &ptrOUT[offsetOUT_3], r3);
            svst1(p8_all, &ptrOUT[offsetOUT_2], r2);
            case 2 :
                svst1(p8_all, &ptrOUT[offsetOUT_1], r1);
                case 1 :
                    svst1(p8_all, ptrOUT, r0);
                }
    }
    ptrOUT += vl;
}

nb_st_elems -= 4;

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).

With the matrix dimensions a multiple of 32, and element size being 8-bit, where vector lengths are 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).

This example code is an SVE implementation of a matrix multiply results calculation:
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) {
        ptrIN_left = &inLeft_MOD[x*K];
        acc0 = svdup_u32(0);
        acc1 = svdup_u32(0);
        acc2 = svdup_u32(0);
        acc3 = svdup_u32(0);
        acc4 = svdup_u32(0);
        acc5 = svdup_u32(0);
        acc6 = svdup_u32(0);
        acc7 = svdup_u32(0);
        for (z=0; z<K; z+=4) {
            svuint8_t a0123 = svld1rq(p8_all, ptrIN_left);
            svuint8_t a4567 = svld1rq(p8_all, &ptrIN_left[16]);
            svuint8_t b_vec = svld1(p8_all, ptrIN_right);
            acc0 = svdot_lane(acc0, b_vec, a0123, 0);
            acc1 = svdot_lane(acc1, b_vec, a0123, 1);
            acc2 = svdot_lane(acc2, b_vec, a0123, 2);
            acc3 = svdot_lane(acc3, b_vec, a0123, 3);
            acc4 = svdot_lane(acc4, b_vec, a4567, 0);
            acc5 = svdot_lane(acc5, b_vec, a4567, 1);
            acc6 = svdot_lane(acc6, b_vec, a4567, 2);
            acc7 = svdot_lane(acc7, b_vec, a4567, 3);
            ptrIN_left += 32;
            ptrIN_right += vl;
        }
        svst1(p8_all, ptrOUT, acc0);
        svst1(p8_all, &ptrOUT[offsetOUT_1], acc1);
        svst1(p8_all, &ptrOUT[offsetOUT_2], acc2);
        svst1(p8_all, &ptrOUT[offsetOUT_3], acc3);
        svst1(p8_all, &ptrOUT[offsetOUT_4], acc4);
        svst1(p8_all, &ptrOUT[offsetOUT_5], acc5);
        svst1(p8_all, &ptrOUT[offsetOUT_6], acc6);
        svst1(p8_all, &ptrOUT[offsetOUT_7], acc7);
    }
    ptrOUT += vl_4;
}

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:
\[
\]

The computation is implemented using the unsigned fixed-point dot-product, by indexed elements UD10T 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).

As a result of 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 4-65
4.4.2 Matrix Multiplication with Real HP Floating-Point Elements on page 4-66
4.4.3 Matrix Multiplication with Real Fixed-Point 8-Bit Input Elements and Real Fixed-Point 32-Bit Output Elements on page 4-70
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 for Linux 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) on page 5-77.
• 5.2 Get Started on page 5-79.
• 5.3 Analyze Emulated SVE on Existing Arm®v8-A Hardware on page 5-81.
• 5.4 View the drrun command on page 5-86.
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 requiring simulators with high overheads. This approach favors faster application execution time, over performance accuracy (for example, ArmIE does not provide any timing information). Faster application execution time allows you to run larger, more realistic applications, coupled with dynamic binary instrumentation.

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. ArmIE and DynamoRIO offer a wide range of instrumentation and metrics gathering tools. ArmIE 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 used 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 with each another using the emulator API. You can see additional
information on the clients, how ArmIE works, and details on how to get started with ArmIE in the
documentation that is provided with the tool, and on the ArmIE web pages.

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

This task describes 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 Linux. Ensure that you have installed the Arm Compiler, and loaded the environment module before beginning this tutorial. For installation and configuration instructions, see Installing Arm Compiler for Linux.

——— 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.

The example provided uses:

- Arm Compiler version 19.3
- ArmIE version 19.2
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 Arm®v8-A Hardware on page 5-81.

Related information
contact Arm Support
5.3 Analyze Emulated SVE on Existing Arm\textsuperscript{v8}-A Hardware

This topic provides an overview of the Arm Instruction Emulator, describes its structure, the clients it uses, and describes how to use it. The topic also describes the kind of analysis and studies that can be performed with the provided clients.

\begin{Verbatim}
Note
\end{Verbatim}

This topic is formed of extracts from a blog. To read the full blog, see \textit{Emulating SVE on existing Arm\textsuperscript{v8}-A hardware using DynamoRIO and ArmIE}.

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 \textit{DynamoRIO dynamic binary instrumentation tool platform} (DBI). ArmIE allows developers to use \textit{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 that you understand the basic principles of instrumenting binaries using the DynamoRIO API. See DynamoRIO's \textit{API Usage Tutorial}.

\subsection*{Overview}

In this topic, the \textit{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 (\texttt{armclang++}) and adds the necessary SVE flag (\texttt{-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 \texttt{main.cpp} file. To reduce the execution time for the evaluation presented here (~12x reduction), the number of iterations (\texttt{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 time.

\subsection*{Instruction Count}

Start with the instruction count client (inscount) and 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 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.

\begin{Verbatim}
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
\end{Verbatim}

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 \textit{View the drrun command on page 5-86}) 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 \texttt{-s} option. In this case, the \texttt{-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. libsvs_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 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

<table>
<thead>
<tr>
<th>Vector length</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
<th>1024-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE utilization</td>
<td>93.43%</td>
<td>89.61%</td>
<td>83.00%</td>
<td>72.49%</td>
</tr>
</tbody>
</table>

As the total number of SVE instructions reduces, the vectors become wider. 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:

```
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
5833081 : bcond
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:

### 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 shows 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
```

4159090 : 0xa54844c9b
4159090 : 0xa5484479
4159090 : 0xa5484458
4159090 : 0xa5484437
4159090 : 0x65b9033a
```

4159090 : 0xa54844c9b
4159090 : 0xa5484479
4159090 : 0xa5484458
4159090 : 0xa5484437
4159090 : 0x65b9033a
```

4159090 : 0xa54844c9b
4159090 : 0xa5484479
4159090 : 0xa5484458
4159090 : 0xa5484437
4159090 : 0x65b9033a
```

77 unique emulated instructions written to undecoded.txt
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 extra 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:
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. Often, these marker instructions should be at the start and end of 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 prevent vectorization.

For 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:

```c
#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, Arm recommends merging both the non-SVE and SVE trace files into a single file. Files can be merged with a simple script that parses the separate memory trace files and orders them into a single full trace output using 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, 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. The merged memory trace snippet of HACCKernels shows the use of colons and commas:

```
Format: <sequence number>: <TID>, <isBundle>, <isWrite>, <data size>, <data address>, <PC>
....
```
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 often used for different types of post-processing analysis. This can include a wide-range of scripts and tools, ranging from simple parsing scripts to more complex cache simulators, and so on. Processing memory traces is outside of the scope of ArmIE, and therefore, no extra tools are currently included.

Some simple scripting experiments that can be done to the fully-merged memory traces:

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

The scripting 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-app does not present a single gather/scatter operation and that load operations dominate the memory accesses performed.
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 Analyse 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

   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

   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

   i         a[i]     b[i]     c[i]
   ===========
   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 the instructions that are executed by example.

3. The libinscount_emulated.so client has an option -only_from_app which only counts the instructions that are 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

   i         a[i]     b[i]     c[i]
   ===========
   1022       165      234      69
   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.
Related tasks
5.2 Get Started on page 5-79

Related information
Building customized analysis instrumentation
Arm Instruction Emulator Arm Developer web page
Analyzing SVE programs with ArmIE