Porting and Optimizing HPC Applications for Arm® SVE

Version 4.0

Documentation

Non-Confidential

Issue 04

101726_4.0_04_en

Porting and Optimizing HPC Applications for Arm® SVE

Documentation

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	17 May 2019	Non-Confidential	First release.
0110-00	5 June 2019	Non-Confidential	Document update to version 1.1.
0200-00	11 November 2019	Non-Confidential	Document update to version 2.0.
0210-00	5 June 2020	Non-Confidential	Document update to version 2.1.
0300-00	30 March 2021	Non-Confidential	Document update to version 3.0.
0400-00	24 August 2021	Non-Confidential	Document update to version 4.0.
0400-01	26 August 2021	Non-Confidential	Documentation update 1 for version 4.0.
0400-02	4 March 2022	Non-Confidential	Documentation update 2 for version 4.0.
0400-03	25 May 2022	Non-Confidential	Documentation update 3 for version 4.0.
0400-04	24 March 2023	Non-Confidential	Documentation update 4 for version 4.0.

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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 affiliates) 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 https://www.arm.com/company/policies/trademarks.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

List of Figures

List of Tables

1. Introduction

Describes how to port your High Performance Computing (HPC) applications to SVE-based Arm hardware, how to start optimizing the applications after they are ported, and what tools Arm provides that can help.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
⚠ Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Learn about 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.

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 Arm®v8-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, provided 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.

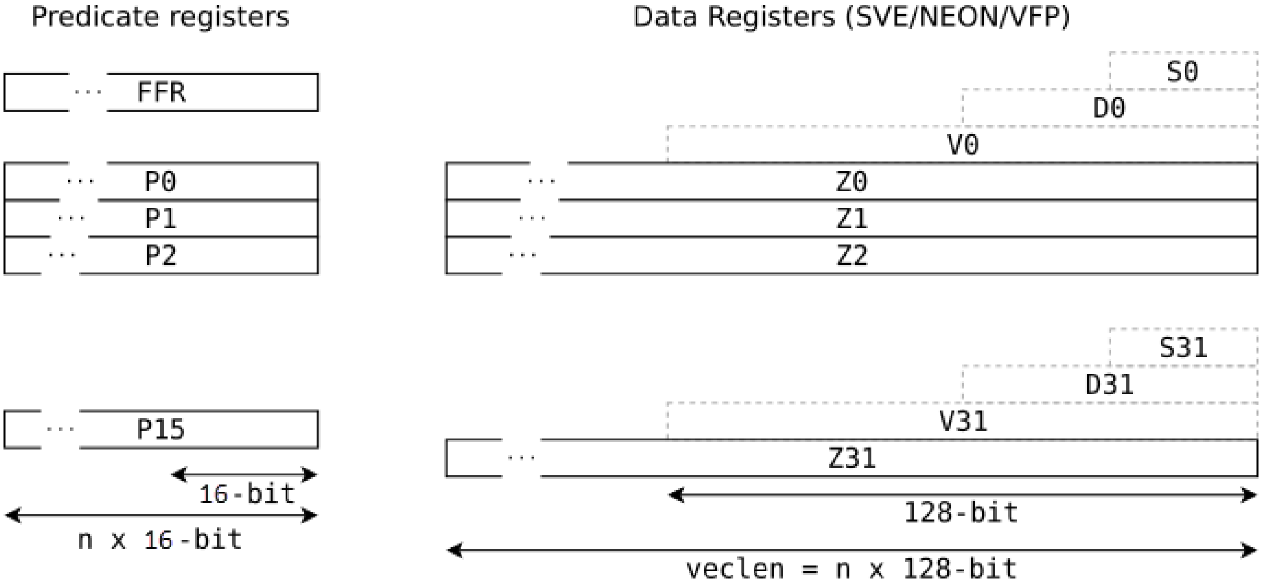
Registers

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:

Figure 2-1: Register overlapping.



`P` registers are 'predicate' registers, which are unique to SVE, and hold one bit for each byte available in a `Z` register. For example, an implementation providing 1024-bit `Z` 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 is either 'active' or 'inactive', depending on the value of its least significant bit.

Note

A one bit predicate lane corresponds to 8-bit bytes in the data register. 8-bits per predicate lane corresponds to 64-bit words in the data register.

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.

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.

Want to evaluate SVE without SVE-enabled hardware?

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

Related information

What is the Scalable Vector Extension? on page 13

2.3 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 the Optimizing HPCG for Arm SVE blog available on the Arm Community website.

The blog was published on June 19, 2019 and used the 19.0 version of the Arm® Compiler for Linux (then named Arm Compiler for HPC) tools. There have been several releases of Arm Compiler for Linux since the 19.0 release.

The case study is included to demonstrate the methodology used, not to present potential performance benefits.

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.

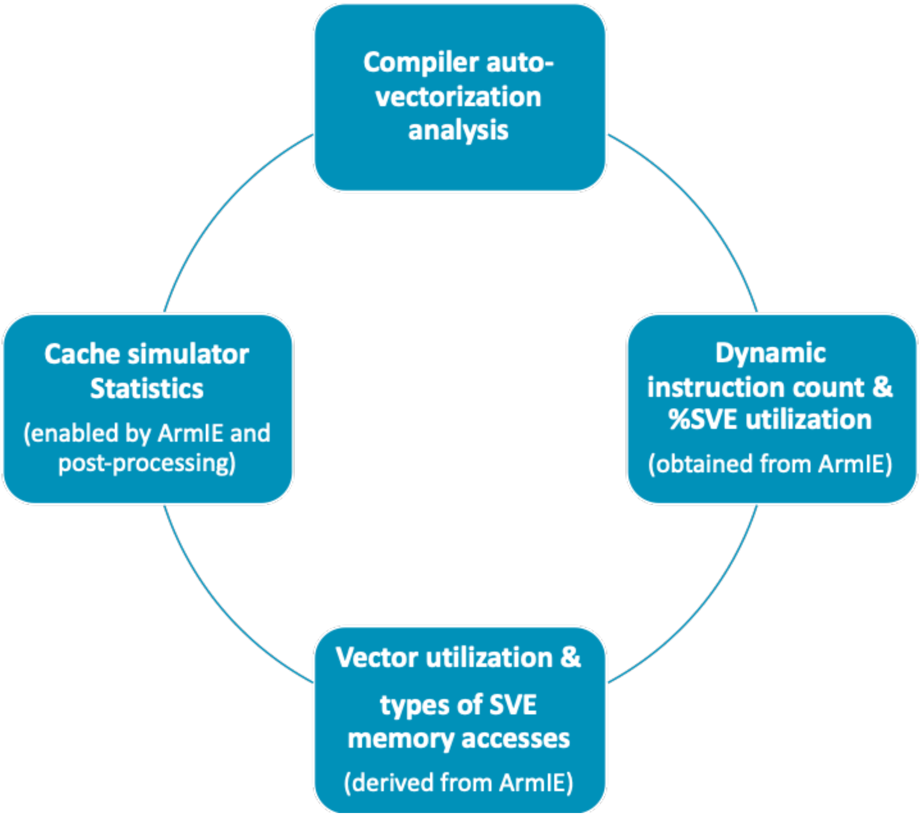
ArmIE comes with four SVE-ready instrumentation clients:

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

You can find further information about how ArmIE works, and how to use these clients in Analyze Emulated SVE on Existing Armv8-A Hardware.

The Arm SVE methodology

Figure 2-2: SVE optimization methodology


Compiler auto-vectorization analysis
Cache simulator Statistics
(enabled by ArmIE and post-processing)
Dynamic instruction count & %SVE utilization
(obtained from ArmIE)
Vector utilization & types of SVE memory accesses
(derived from ArmIE)

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.

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:

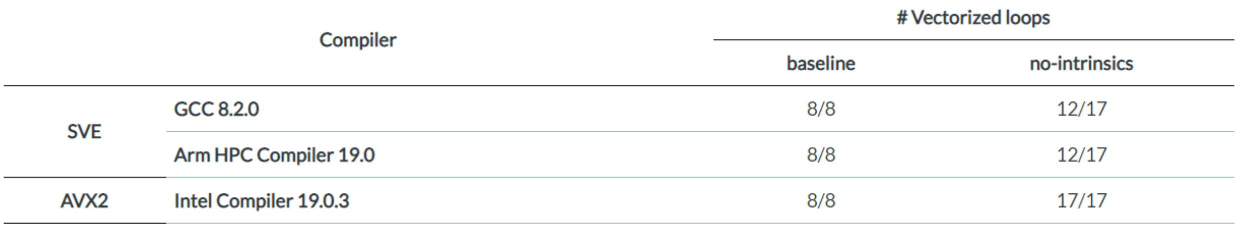
1. Baseline (reference HPCG code)
2. No-intrinsics (optimized HPCG code without SVE intrinsics. For more information about this code, see Parallelizing HPCG).

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.

Figure 2-3: Compiler autovectorization comparison



Compiler		# Vectorized loops	
		baseline	no-intrinsics
SVE	GCC 8.2.0	8/8	12/17
	Arm HPC Compiler 19.0	8/8	12/17
AVX2	Intel Compiler 19.0.3	8/8	17/17

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.

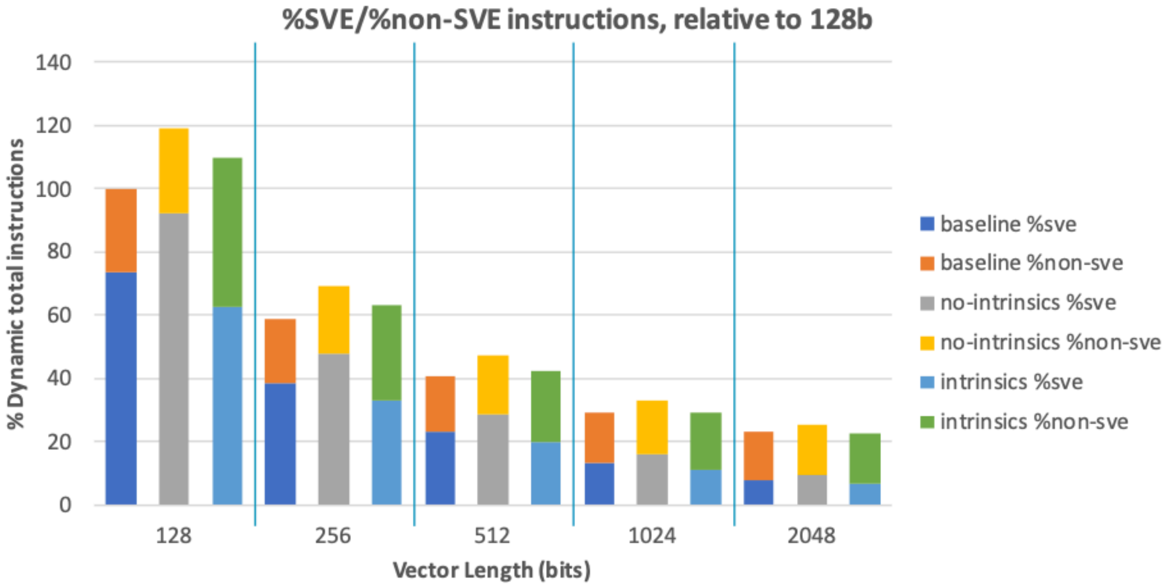
Instruction counts

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.

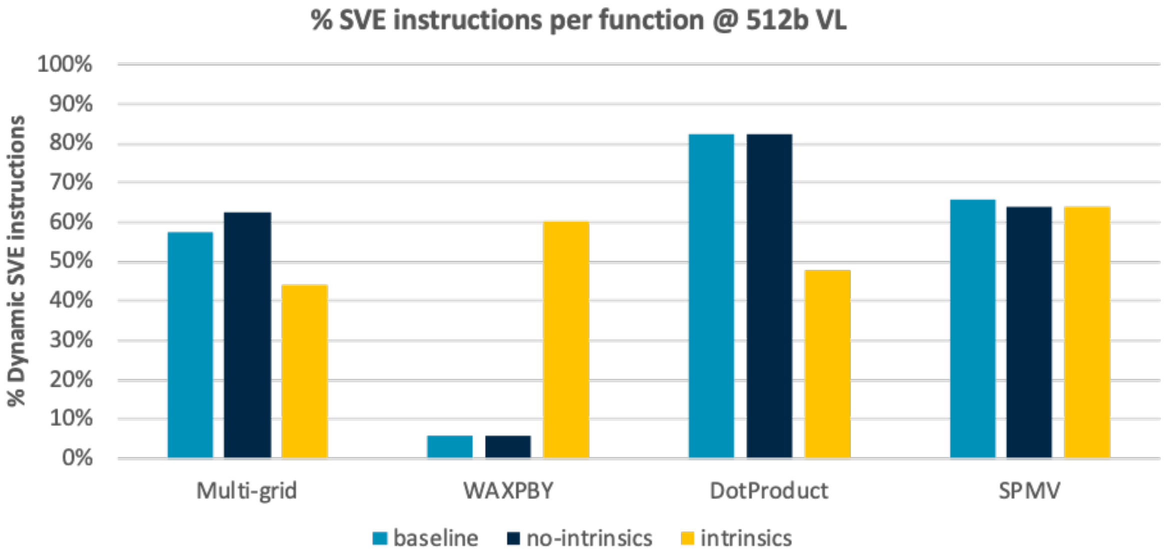
Figure 2-4: Instruction count reduction when increasing vector length



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:

Figure 2-5: Percentage of SVE instructions per computational kernel



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.

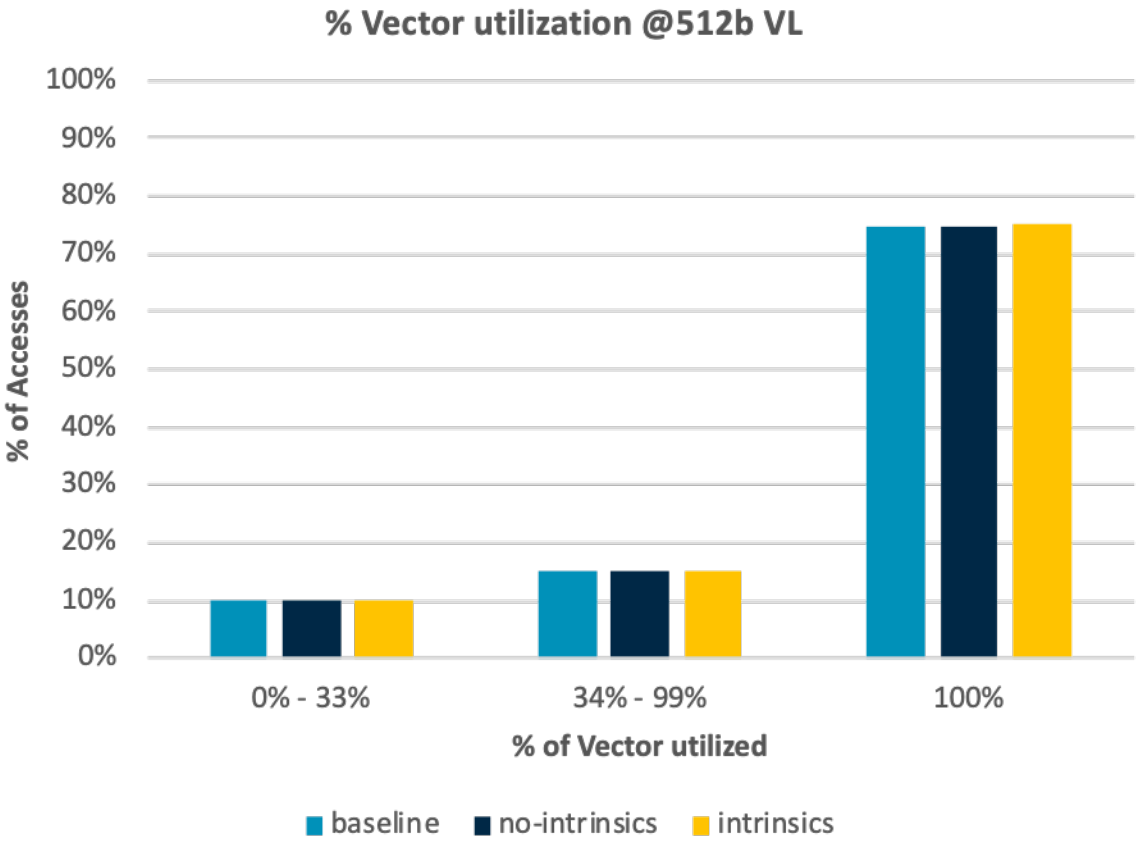
Vector lane utilization and Memory accesses breakdown

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

Figure 2-6: Average lane utilization



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.

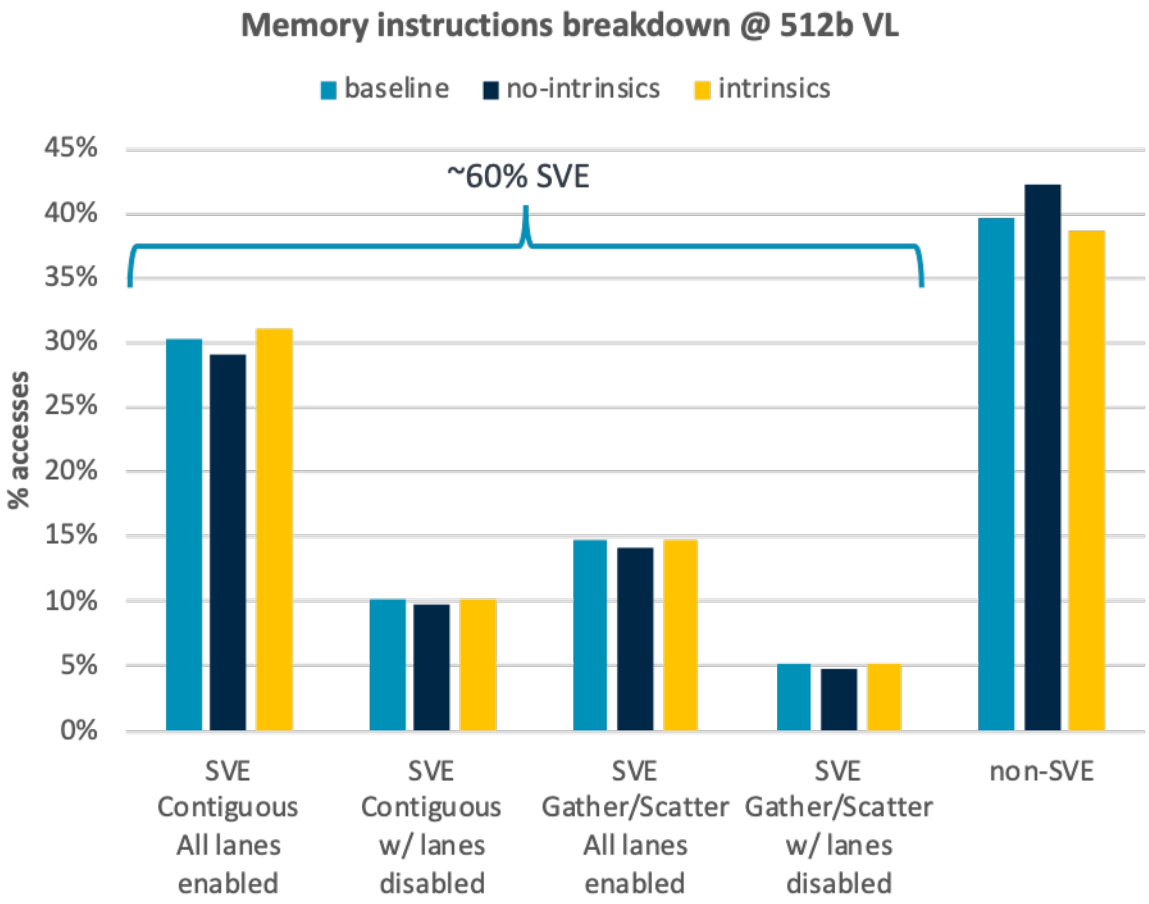
Assuming a similar vector utilization for non-memory SVE operations, you can infer that the vectors are fully utilized most of the time, averaging a vector lane utilization of ~82% for all HPCG versions.

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

Figure 2-7: Memory accesses breakdown



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.

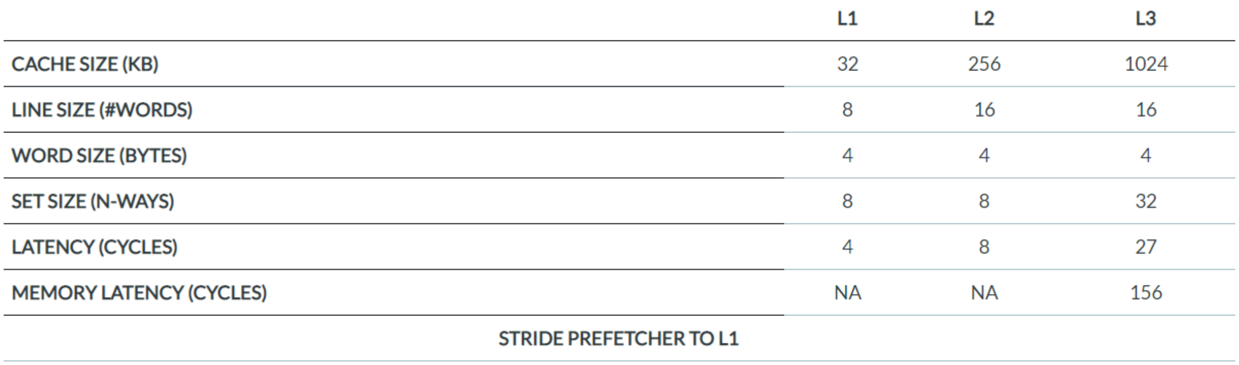
The chart also distinguishes between SVE memory accesses with all lanes active, or some of the lanes disabled. Around 60% of the memory accesses are generated by SVE memory instructions, with half of those being contiguous accesses with all lanes enabled. SVE gather-load and scatter-store accesses represent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

Figure 2-8: Simulator parameters

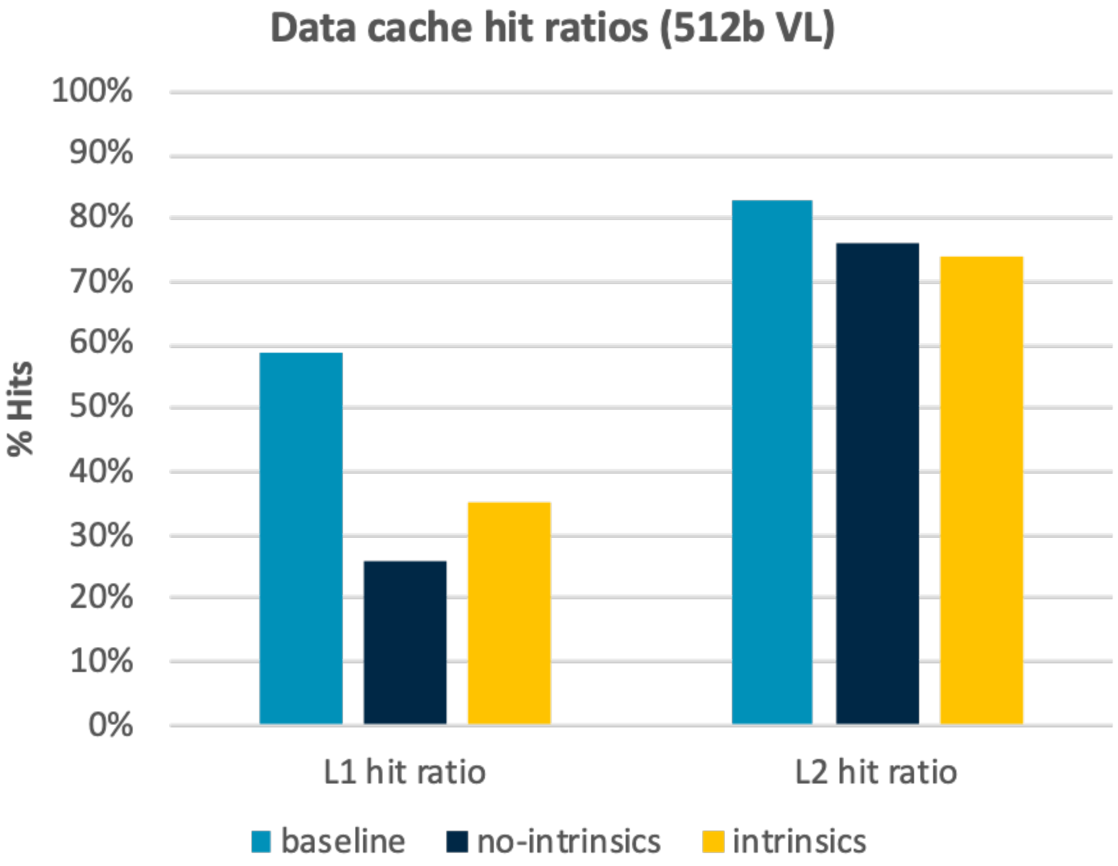


	L1	L2	L3
CACHE SIZE (KB)	32	256	1024
LINE SIZE (#WORDS)	8	16	16
WORD SIZE (BYTES)	4	4	4
SET SIZE (N-WAYS)	8	8	32
LATENCY (CYCLES)	4	8	27
MEMORY LATENCY (CYCLES)	NA	NA	156
STRIDE PREFETCHER TO L1			

Result

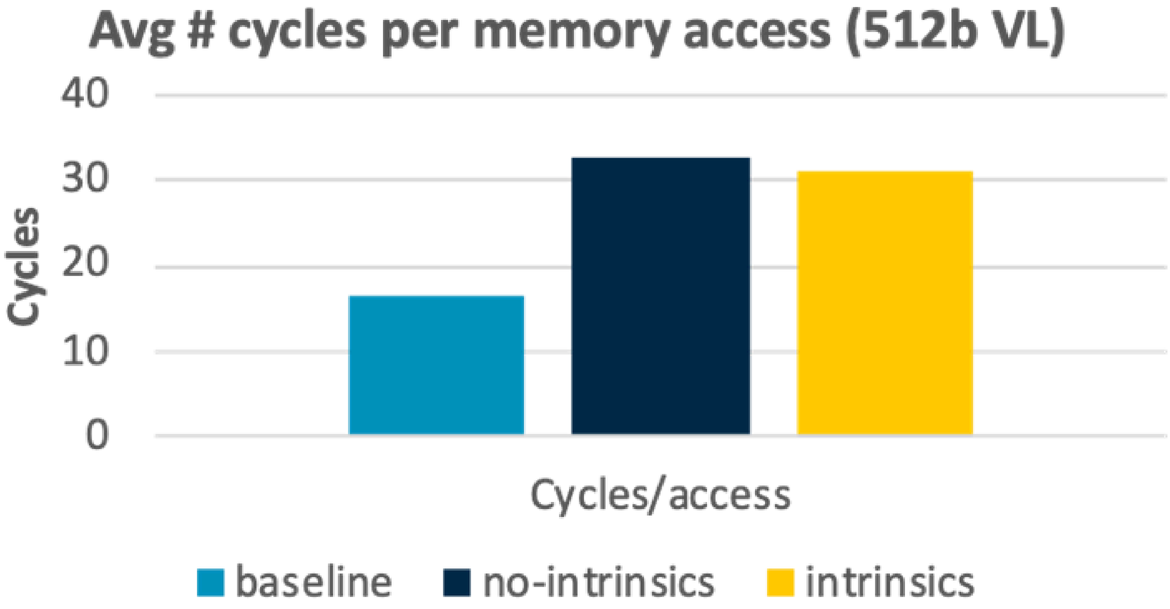
When implementing the parallelization techniques to optimize the reference HPCG code, there is a potential cache hit ratio degradation. This behavior with the cache simulator is observed and reflected in the chart below. Interestingly, the hit ratio is improved in the intrinsics version, when compared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

Figure 2-9: Simulated cache hit ratio



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:

Figure 2-10: Average number of cycles per memory access

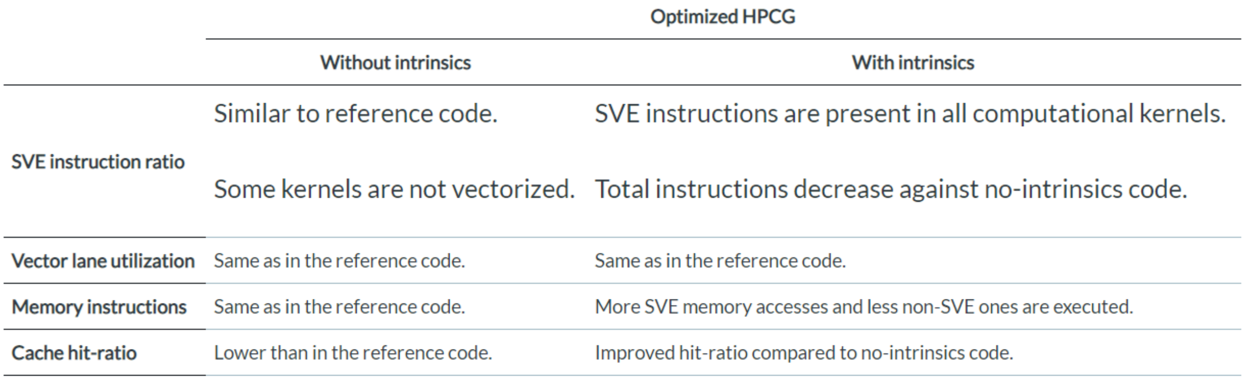


Putting everything together

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:

Figure 2-11: Optimization summary



	Optimized HPCG	
	Without intrinsics	With intrinsics
SVE instruction ratio	Similar to reference code. Some kernels are not vectorized.	SVE instructions are present in all computational kernels. Total instructions decrease against no-intrinsics code.
Vector lane utilization	Same as in the reference code.	Same as in the reference code.
Memory instructions	Same as in the reference code.	More SVE memory accesses and less non-SVE ones are executed.
Cache hit-ratio	Lower than in the reference code.	Improved hit-ratio compared to no-intrinsics code.

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

3. Port and optimize your application to SVE-enabled Arm-based processors

This chapter describes the tools available, and the steps to take, to help you port and optimize your applications for an Scalable Vector Extension (SVE)-enabled Arm-based processor.

Most applications port to the Arm architecture with little or no modification, because:

- All major Linux distributions support Arm. Major Linux distribution support provides an extensive library of common Linux packages that are built for AArch64.
- Applications and dependencies can be recompiled using compilers that support AArch64 applications for Linux user space.
- Arm® Compiler for Linux is available.
- GNU Compiler Collection (GCC) is fully supported.

Note

Arm C/C++/Fortran Compiler also accepts GCC compiler options, where possible.

However, there are a few features of the Arm architecture that might impact your application. These features are described in the **Troubleshooting** section in the Port your application topic.

3.1 Tools to Port and Optimize

Arm provides tools to help you port and optimize applications for Arm.

Arm Compiler for Linux

Arm® Compiler for Linux is comprised of Arm C/C++/Fortran Compiler and Arm Performance Libraries.

Arm C/C++/Fortran Compiler

Arm C/C++/Fortran Compiler is a Linux user space, C/C++, and Fortran compiler, tuned for scientific computing, HPC, and enterprise workloads. The compiler offers:

- Processor-specific optimizations for various server-class Arm-based platforms.
- Optimal shared-memory parallelism using latest Arm-optimized OpenMP run-time.
- Optimized scalar and vector maths functions.
- C++17 and Fortran 2003 language support (partial Fortran 2008 support) with OpenMP 4.5 (OpenMP 5.0 for C/C++).
- Support for Armv8-A, SVE, and SVE2 architecture extensions.
- Based on leading open-source compiler projects: LLVM, Clang, and Flang.

- Runs on leading Linux distributions: Red Hat, SLES, and Ubuntu.

Arm Performance Libraries

Arm Performance Libraries is a 64-bit, Armv8-A core math libraries, optimized for Server and HPC applications on Arm-based platforms, providing best-in-class serial and parallel performance. The libraries offer:

- Commonly used low-level math routines: BLAS, LAPACK, FFT, sparse linear algebra, and libamath functions.
- An FFTW-compatible interface for FFT routines.
- Batched BLAS support.
- Generic Armv8-A optimizations by Arm.
- Tuning for specific platforms in collaboration with silicon vendors.
- Validated with NAG's industry standard test suite.
- Available for Arm C/C++/Fortran Compiler and GCC.

Resources

- For the latest information and other resources, see the Arm Compiler for Linux Developer web page.
- Arm C/C++ Compiler Developer and Reference guide
- Arm Fortran Compiler Developer and Reference guide
- Arm Performance Libraries Developer and Reference guide

Arm Forge

A cross-platform toolkit to debug (Arm® DDT), profile (Arm® MAP), and analyze (Arm® Performance Reports) high-performance parallel applications. Arm Forge:

- Is available on most of the Top500 machines in the world.
- Is fully supported by Arm on x86, ppc64le, Nvidia GPUs, and AArch64.
- Has powerful and in-depth error detection mechanisms (including memory debugging).
- Identifies and understands bottlenecks using a sampling-based profiler.
- Available at any scale (from the desktop to leadership-class HPC).
- Has supports for remote interactive sessions.
- Analyzes metrics around CPU, memory, I/O, and hardware counters.
- Supports custom metrics.
- Provides simple guidance on how to improve workload efficiency.
- Can be integrated into various systems for validation (for example, continuous integration).
- Can be automated to remove the requirement for user intervention.

Resources

- For the latest release information and other resources, see the Arm Forge Developer web page.

- For Arm DDT documentation, see Arm DDT User Guide.
- For Arm MAP documentation, see Arm MAP User Guide.
- For Arm Performance Reports documentation, see Arm Performance Reports User Guide.

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 on non-SVE Armv8-A hardware. Based on the DynamoRIO dynamic binary instrumentation framework, ArmIE enables the customized instrumentation of SVE binaries, which enables you to analyze specific aspects of runtime behavior.

Note

ArmIE does not provide any timing information.

ArmIE:

- Emulates instructions by converting them to native Armv8-A instructions.
- Is free to use, you do not need a license.
- Is available on several Linux distributions, including Ubuntu, RHEL, and SLES.

Resources

- For more information about Arm Instruction Emulator, see Emulate your application with Arm Instruction Emulator
- For the latest release information and to download Arm Instruction Emulator, see the Arm Instruction Emulator webpage.
- To find out where to learn more about the Scalable Vector Extension (SVE), see Coding for SVE resources

3.2 Port your application

This high-level topic describes how to port application code that was written for non-Arm architectures, to run on SVE-enabled Arm-based hardware (or emulation).

Note

If you are porting code that was previously optimized for Neon®, read Migrate from Neon to SVE, which is tailored to this type of code migration.

Procedure

1. Port your application dependencies.

Note

When building Fortran applications, to ensure that the Fortran interface is compatible with your application, you must compile the application dependencies with the same toolchain that you use to compile your application.

Use of external libraries is increasingly common, and a conscious design choice for many projects. Common dependencies include:

- **I/O libraries**

  For example, HDF5 and NetCDF (C, Fortran, and parallel flavors).

- **Maths libraries and toolkits**

  For example PETSc, HYPRE, Trilinos, ScaLAPACK, LAPACK, and BLAS.

  Note

  Arm® Performance Libraries provides optimized LAPACK, BLAS, sparse linear algebra, and libamath implementations.

- **Fast Fourier Transforms**

  For example, FFTW.

  Note

  Arm Performance Libraries provides an FFT implementation which is compatible with FFTW's interface.

- **Communication layers, or execution environments**

  For example, Open MPI, OpenUCX, and Charm++.

- **Libraries providing performance portability and memory abstraction**

  For example, Kokkos and RAJA.

Often, these dependencies have been built on Arm before with the Arm and GNU toolchains. For a full list of ported applications, see the community-driven Packages Wiki.

2. Update your compiler.

   During your build configuration, specify which C, C++, and Fortran compilers to use. For example, for Arm Compiler for Linux, set:

   ```
   CC=armclang
   CXX=armclang++
   FC=armflang
   F77=armflang
   ```

For GCC, set:

```
CC=gcc
CXX=g++
FC=gfortran
F77=gfortran
```

For MPI builds you might need to use the MPI wrappers. These are usually the same for all compilers:

```
CC=mpicc
CXX=mpicxx
FC=mpifort
```

3. Update your compiler options.

   Most GCC options are supported by Arm Compiler for Linux. If you are compiling and running your application on the same SVE-enabled hardware, Arm recommends that you compile with the `-mcpu=native` option (in addition to your other compile options). `-mcpu=native` ensures that your compiler uses the micro-architectural optimizations suitable for your system. To enable the optimal version of Arm Performance Libraries for your target, also include `-armpl` (for Arm Compiler for Linux) or `-larmpl` (for GCC) in your compile and link commands.

   Note

   If you are compiling and running on different machines, read Accessing the library to find out which compiler options you can use for Arm Performance Libraries.

4. Build your application as you would normally.

   If you find any issues with your build, see the **Porting - Troubleshooting** section below.

5. Run your test suite.

   If you do not have access to SVE-enabled Arm-based hardware, you can use Arm Instruction Emulator to run your binary and emulate the SVE instructions.

   Warning

   Regression tests that rely on bit-wise identical answers might not be portable between architectures.

Porting - troubleshooting

Here are some problems that you might find when porting your application:

- *Configure is unable to identify your platform*

  Configure might not be able to identify your platform because the `config.guess` supplied with the application is out of date. This can also be true for a `config.guess` already installed on your system and used by some configure scripts.

Solution

To fix this problem, obtain up-to-date versions:

```
wget 'http://git.savannah.org/gitweb/?
p=config.git;a=blob_plain;f=config.guess;hb=HEAD' -O config.guess
wget 'http://git.savannah.gnu.org/gitweb/?
p=config.git;a=blob_plain;f=config.sub;hb=HEAD' -O config.sub
```

- *Libtool fails to link Fortran applications or interfaces*

  Libtool does not recognize Arm C/C++/Fortran Compiler as a Fortran compiler. Therefore, it is unable to set the correct flags for linking the binary.

  #### Solution

  Ensure that Libtool uses the correct compiler options with Arm Compiler for Linux by modifying it after running configure:

  ```
  sed -i -e 's#wl=""#wl="-Wl,"#g' libtool
  sed -i -e 's#pic_flag=""#pic_flag=" -fPIC -DPIC"#g' libtool
  ```

  Some widely used applications and libraries, for example Open MPI, have already incorporated a fix to address this issue at the configure stage.

- *#ifdefs in the makefile are not being set*

  There might be compiler-dependent `#ifdefs` in the source which are not set.

  #### Solution

  You might need to update the source to use the `_FLANG` and `_clang` macros, or manually set existing compiler macros, such as `-D_PGI`.

- *Unsupported language features*

  Your code might be using language features which are not supported by Arm C/C++/Fortran Compiler.

  #### Solution

  Check the support status of the compiler, for:

  - Fortran 2003 standard support.
  - Fortran 2008 standard support (Partial).
  - Fortran OpenMP 4.0 and Fortran OpenMP 4.5 (Partial) support.
  - C/C++ OpenMP 4.0, C/C++ OpenMP 4.5 (Partial), and C/C++ OpenMP 5.0 (Partial) support.

- *You experience a race condition you have not encountered before*

AArch64 uses a weak memory model. A weak memory model is where read and writes can be reordered. Sometimes, it means that explicit memory barriers are needed on AArch64 that were not required on other architectures.

Solution

Implement explicit memory barriers for AArch64. These are described in Barriers in the Arm Cortex-A Series Programmer's guide for Armv8-A, and in Appendix J of the ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.

- *Integer divide by zero*

  On AArch64, an integer divide by zero does not generate an error; instead it returns as zero.

  Note

  This is not the case for floating-point divide by zero.

  Occasionally, an undetected divide by zero might be allowing an application to run erroneously when it should fail.

  Solution

  It might be necessary to explicitly catch attempted divide-by-zeros in software. For example, if you have:

  ```
  c = a / b
  ```

  Test for `b==0` before executing the divide, and generate a warning, or adjust the program flow accordingly.

- *Thread mapping and pinning on Arm*

  Arm chips can have lots of cores. It is important to manage how your threads get mapped to the cores, and how they are pinned.

  Solution

  Map your threads to cores using the available mapping devices:

  - OpenMP environment variables
  - OpenMPI run flags
  - Numactl

- *Segmentation fault when calling an Arm Performance Libraries function*

  Segmentation faults can occur when you are linking against the wrong version of the library with either 32-bit integers or 64-bit integers.

Solution

Compile and link for 32-bit integers (`-armpl=lp64`) or for 64-bit integers (`-armpl=ilp64`), as required.

- *Building Position Independent Code (PIC) on AArch64*

  Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lowercase `-fpic` compiler flag with GCC compilers (gfortran, gcc, g++), instead of using the uppercase `-fPIC` flag.

  Note

  - This issue does not occur when using the `-fpic` flag with Arm Compiler for Linux (armflang/armclang/armclang++), and it also does not occur on x86_64 because `-fpic` operates the same as `-fPIC`.
  - PIC is code which is suitable for shared libraries.

  ### Cause

  Using the `-fpic` compiler flag with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and can provide code size and performance benefits. However, it also sets a limit of 32k for the Global Offset Table (GOT), and the build can fail at the executable linking stage because the GOT overflows.

  Note

  When building PIC with Arm Compiler for Linux on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

  ### Solution

  Consider using the `-fPIC` compiler flag with GCC compilers on AArch64, because it ensures that the size of the GOT for a dynamically linked executable are large enough to allow the entries to be resolved by the dynamic loader.

  To increase code portability, Arm recommends using `-fPIC` (instead of `-fpic`) when compiling with Arm C/C++/Fortran Compiler.

  For more information, see Building Position Independent Code (PIC) on AArch64.

- *Applications supporting GCC builds on Arm - but use Armv-7 compiler flags*

  Some Armv7 flags that are needed for Armv7, cause errors for Armv8 targets. For example, on Armv8 Neon is compulsory, so the flag `-fp=neon` does not exist on Armv8. If it is used when compiling for Armv8, GCC does not recognize it and causes an error.

  ### Cause

  Typically, the flags are incorrect in Makefiles.

Solution

Update your Makefiles to only use compatible Armv8 compiler flags.

Related information

Get started on Arm
Packages Wiki
Develop on Arm
Arm Compiler for Linux
Arm Performance Libraries (part of Arm Compiler for Linux)
Arm Performance Libraries (Standalone)

3.3 Optimize

After porting your application to Arm, the next step is to optimize it. The following steps describe how you can use the Arm compilers, debugger, and profiler to enhance the performance of your ported application on Arm.

Note

If you are migrating code that was previously optimized for Neon®, you can find more specific instructions about migrating highly-optimized Neon code to SVE code in Migrate from Neon to SVE

Procedure

1. Optimize your code using compiler optimization options:
   - Enable auto-vectorization with the `-O<level>` options:

Table 3-1: Arm Compiler for Linux optimization options

Option	Description	Auto-vectorization
`-O0`	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.	Never
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.	Disabled by default.

Option	Description	Auto-vectorization
`-O2`	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.	Enabled by default.
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.	Enabled by default.
`-Ofast`	Enable all the optimizations from level 3, including those performed with the `–ffp-mode=fast armclang` option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.	Enabled by default.

Note

- The `-Ofast` option enables all the optimizations from `-O3`, but also performs other aggressive optimizations that might violate strict compliance with language standards. If your Fortran application has issues with `-Ofast`, to force automatic arrays on the heap, try `-Ofast –fno-stack-arrays`.
- If `-Ofast` is not acceptable and produces the wrong results because of the reordering of math operations, use `-O3 -ffp-contract=fast`. If `–ffp-contract=fast` does not produce the correct results, then use `-O3`.

For a more detailed description of auto-vectorizing your code for Arm Neon, see Migrate from Neon to SVE.

- For C/C++ and Fortran code, the vectorized implementations of the standard math library functions are being used as needed.

Note

Vector implementations are used, when possible, by default, but can be disabled using the `-fno-simdmath` compiler flag.

- Optimize for your hardware. To compile your code for your specific processor, use the `–mcpu=<target>` option. Compiling for the specific processor enables the compile to optimize knowing the architecture and microarchitectural versions implemented on that processor.

In summary, if you will compile and run your application on the same machine, a typical set of compiler optimization options are:

```
{armclang|armclang++|armflang} -fsimdmath -mcpu=native -c -O3
 <source>{.c|.cpp|.f}
```

For a full list of compiler optimization options, see the options descriptions in the Arm C/C++ Compiler Reference Guide or the options descriptions in the Arm Fortran Compiler Reference Guide.

2. Enable optimized Arm Performance Libraries functions.

   Arm Performance Libraries provide optimized standard core math libraries for high-performance computing applications on Arm processors:

   - BLAS: Basic Linear Algebra Subprograms (including XBLAS, the extended precision BLAS).
   - LAPACK: A comprehensive package of higher level linear algebra routines. To find out what the latest version of LAPACK that is supported in Arm Performance Libraries is, see Arm Performance Libraries.
   - FFT functions: a set of Fast Fourier Transform routines for real and complex data using the FFTW interface.
   - Sparse linear algebra.
   - libamath: a subset of libm, which is a set of optimized mathematical functions.
   - libastring: A subset of libc, which is a set of optimized string functions.

   Arm Performance Libraries also provides improved Fortran math intrinsics with auto-vectorization. For more information about the supported Fortran intrinsics in Arm Fortran Compiler see the Arm Fortran Compiler Developer and Reference Guide

   To instruct your compiler to load the optimum version of Arm Performance Libraries for your target architecture and implementation, use the `-armpl` (for Arm Compiler for Linux) or `-larmpl` (for GCC) option.

   For Arm Compiler for Linux, the `-armpl` option enables the compiler to use optimized versions of the mathematical functions, and to use the appropriate Arm Performance Libraries header files (during compilation) and libraries (during linking). For `armclang|armclang++`, `-armpl` also enables the compiler to auto-vectorize mathematical functions. For `armflang`, auto-vectorization of mathematical functions is enabled by default.

   To obtain the best performance enhancements for your build and run environments, combine `-armpl` with the `-mcpu` or `-march` options, as appropriate. For example, use `-mcpu=native` when compiling your application on the same system that you will run it on:

```
{armclang|armclang++|armflang} -mcpu=native -armpl=<arg1>,<arg2>... <options>
 code_with_math_routines{.c|.cpp|.f*}
```

Note

- If your build process compiles and links as two separate steps, ensure that you add the same `-armpl` and `-mcpu|-march` options to both.
- If you are not compiling and linking on the same target as you will run the binary on, or you are building a binary to run with emulation software, read Accessing the library to learn more about the options to use with `-armpl`.
- For GCC, you need to load the correct environment module for the system and explicitly link to your chosen flavor (lp64/ilp64, mp) with full library path:

  ```
  {gcc|gfortran} <options> code_with_math_routines{.c|.f*} -larmpl
  ```

To learn more about using Arm Performance Libraries, read Get Started with Arm Performance Libraries (part of Arm Compiler for Linux) or Get Started with Arm Performance Libraries (Standalone)

3. Use Arm Optimization Reports.

   Arm Optimization Report is a feature of Arm Compiler for Linux that builds upon the llvm-opt-report tool available in open-source LLVM. The new Arm Optimization Report feature makes it easier to see what optimization decisions the compiler is making about unrolling, vectorizing, and interleaving, all in-line with your source code.

   To enable Arm Optimization Reports:

   a. At compile time, add the `-fsave-optimization-record` to the command line.

      A `<filename>.opt.yaml` report is generated by the compiler, where `<filename>` is the name of the binary.

   b. Use Arm Optimization Report (`arm-opt-report`) to inspect the `<filename>.opt.yaml` report as augmented source code:

      ```
      arm-opt-report <filename>.opt.yaml
      ```

      The annotated source code appears in the terminal.

   For more information, see the C/C++ Arm Optimization Reports documentation or the Fortran Arm Optimization Reports documentation.

4. Use Arm C/C++/Fortran Compiler Optimization Remarks.

   Optimization Remarks is another took that provides you with information about the choices that are made by the compiler. Optimization Remarks can be used to see which code has been inlined or to understand why a loop has not been vectorized.

   To enable Optimization Remarks, pass one or more of the following `-Rpass` flags at compile time:

Table 3-2: -Rpass flags to enable optimization remarks

-Rpass flags	Description
`-Rpass=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has optimized.
`-Rpass-analysis=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has analyzed.
`-Rpass-missed=<regexp>`	To request information about what Arm C/C++/Fortran Compiler failed to optimize.

In each case, `<regexp>` is used to select the type of remarks to provide. For example, loop-vectorize for information on vectorization, and inline for information on in-lining, or `.*` to report all Optimization Remarks. `Rpass` accepts regular expressions, so (loop-vectorize|inline) can be used to capture any remark on vectorization or inlining.

For example, to get actionable information on which loops can and cannot be vectorized at compile time, pass:

```
-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize
 -g
```

Note

- Optimization Remarks are only available when you have set an appropriate debug flag, for example `-g`.
- Optimization Remarks are piped to `stdout` at compile time.

For more information, see the Optimization Remarks documentation for C/C++ or for Fortran.

5. If your code is Fortran, use the Arm Fortran Compiler directives.

   Arm Fortran Compiler supports general-purpose and OpenMP-specific directives:

   - `!DIR$ IVDEP` - A generic directive to force the compiler to ignore any potential memory dependencies of iterative loops and vectorize the loop.
   - `!$OMP SIMD` - An OpenMP directive to indicate that a loop can be transformed into a SIMD loop.
   - `!$MEM PREFETCH` - Tells the compiler to generate prefetch instructions to fetch elements and load them in the data cache, ahead of their first use
   - `!DIR$ VECTOR ALWAYS` - Forces the compiler to vectorize a loop regardless of any potential performance implications.

     Note

     The loop must be vectorizable.

   - `!DIR$ NO VECTOR` - Disables vectorization of a loop.

- `!DIR$ UNROLL` - Instructs the compiler to unroll the loop it precedes.
- `!DIR$ NOUNROLL` - Instructs the compiler not to unroll the loop it precedes.

For more information, see the directives section of the Arm Fortran Compiler reference guide.

6. If your code is C/C++, use pragmas.

   Arm C/C++ Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas use, and extend, the `pragma clang loop` directives:

   - `#pragma clang loop vectorize(assume_safety)`: Indicates to the compiler that the loop that follows contains no data dependencies between loop iterations that would prevent vectorization. The compiler might be able to use this information to vectorize a loop, where it would not typically be possible.
   - `#pragma clang loop vectorize(disable)`: Suppresses auto-vectorization on a specific loop.
   - `#pragma clang loop vectorize_width(scalable)`: Suppresses Neon instructions and prefer SVE instructions.
   - `#pragma clang loop vectorize_width(fixed)`: Suppresses SVE instructions and prefer Neon instructions.
   - `#pragma clang loop unroll(enable)`: Unroll scalar loops to the internal limit.
   - `#pragma clang loop unroll_count(\_value\_)`: Unroll scaler loops to a custom value.
   - `#pragma clang loop interleave(enable)`: Unroll vectorizable loops to the internal limit.
   - `#pragma clang loop interleave_count(\_value\_)`: Unroll vectorizable loops to a custom value.

7. Optimize by iteration.

   Use Arm Forge to analyze application performance, as well as debug and profile your ported application.

   - Use Arm Performance Reports to characterize and understand the performance of HPC application runs
   - Use Arm DDT to debug your code to ensure that the application is correct. Arm DDT can be used both in an interactive and non-interactive debugging mode, and optionally, integrated into your CI workflows.
   - Use Arm MAP to profile your code to measure your application performance. To understand the code performance, Arm MAP collects:
     - A broad set of performance metrics.
     - A broad set of time classification metrics.
     - Instruction information (hardware counters).
     - Specific metrics (for example MPI call and message rates, I/O data rates, and energy data).
     - Custom metrics (metrics defined by you).

To optimize application performance, use the Arm Performance Reports, Arm DDT, Arm MAP tools and follow an iterative identification and resolving cycle:

a. Run your code on real workloads and generate a performance report with Arm Performance Reports.

b. Use Arm MAP to examine the I/O and trace and debug suspicious or slow access patterns.

   Common problems include:

   - Checkpointing too often.
   - Many small read and writes.
   - Using your home directory instead of the scratch directory.
   - Multiple nodes using the filesystem a the same time.

c. Use Arm Performance Reports to identify the workload balance of your application, then use Arm MAP to dive into partitioning code.

   Common problems include:

   - Your dataset is too small to efficiently run at scale.
   - I/O contention causes late sender.
   - Partitioning code bugs.

d. Use Arm MAP to identify lines of code that are causing memory access pattern problems.

   Common problems include:

   - Initializing memory on one processor but using it on a different processor.
   - Arrays of structures causing inefficient cache utilization.
   - Caching results when re-computation is more efficient.

e. Use Arm Performance Reports to track communication performance, and Arm Forge to see which communication calls are slow and why.

   Common problems include:

   - Short, high-frequency messages are very sensitive to latency.
   - Too many synchronizations.
   - No overlap between communication and computation.

f. Use Arm Performance Reports to observe the processor utilization and synchronization overhead, then use Arm Forge to identify the corresponding code.

   Common problems include:

   - Implicit thread barriers inside tight loops.
   - significant processor idle time because of workload imbalance.
   - Threads migrating between processors at runtime.

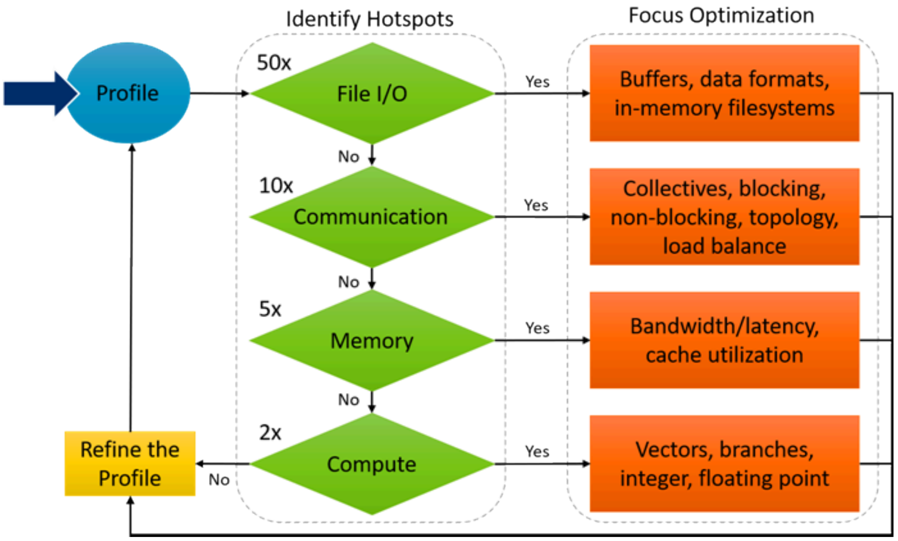
g. Use Arm Performance Reports to observe the numerical intensity and level of vectorization, and Arm Forge to identify the hot loops and unvectorized code.

   Common problems include:

   - Sub-optimal compiler options for your system.
   - Numerically-intensive loops with hard to vectorize patterns.
   - Not utilizing highly-optimized math libraries.

Example slowdown factors that can occur are:

Figure 3-1: Iterative optimization cycle.


Identify Hotspots
Focus Optimization
Profile
50x
File I/O
Yes
Buffers, data formats, in-memory filesystems
No
10x
Communication
Yes
Collectives, blocking, non-blocking, topology, load balance
No
5x
Memory
Yes
Bandwidth/latency, cache utilization
No
2x
Compute
Yes
Vectors, branches, integer, floating point
No
Refine the Profile

Note

The 50x, 10x, 5x, and 2x numbers in the figure are potential slowdown factors that Arm has observed in real-world applications (when that aspect of performance is done incorrectly).

For more information about using analysis, debugging, and profiling tools, see the Arm Forge user guide.

Related information

Packages Wiki

Latest additions to the Armv8-A architecture

Arm Compiler for Linux

Arm Forge

3.4 OpenMP Thread Mapping

Describes what to consider when managing OpenMP thread mapping on AArch64 platforms.

The placement and management of OpenMP threads can have a significant impact on the performance of OpenMP enabled applications. By default, no environment variables are set to control the placement and binding of OpenMP threads. Not controlling the OpenMP threads leaves the kernel free to distribute the threads over the available resources, and swap the OpenMP threads between physical cores dynamically, which is unlikely to be the best-performing configuration for your applications.

With the large physical core-counts on infrastructure-scale AArch64 CPUs, the impact of the kernel swapping OpenMP threads between available cores, a process known as *thread migration*, can influence High Performance Computing (HPC) application performance more than it would on other platforms. More specifically, HPC applications are data driven, so the locality of data and threads is important. Thread migration causes problems because it moves threads away from the caches that have the data in. This is especially problematic for NUMA systems, where the tread is migrated to another NUMA node, and the data must be fetched across NUMA domains.

If your application depends on OpenMP parallelism, it is important to understand:

- How to control and manage OpenMP thread placement.
- How to understand where OpenMP threads are being executed.
- What configuration works best for each application.

With any compiler, Arm recommends using `lstopo` to understand the thread layout of the system. For more information, see the **lstopo** section.

OpenMP environment variables

By default, the following OpenMP environment variables are unset:

Table 3-3: OpenMP environment variables

Environment variable	Description
`OMP_NUM_THREADS=<value>[, <value>[, ...]]`	Specifies the default number of threads to use: • If a single value is passed, your application uses a single level of parallelism. In other words, nested parallelism is disabled. • If a comma-separated list of values are passed, the values denote the number of threads to use at each level of nesting, starting from the outermost parallel region.

Environment variable	Description
`OMP_PROC_BIND={true|false|close|spread|master}`	Specifies whether threads can be moved between processors, and controls how the threads are distributed: • `close`: Implies that threads are bound and laid out in successive order, based on the unit of `OMP_PLACES`. `close` packs OpenMP threads as close to each other as possible, which is good for communication (avoids inter-socket) and cache sharing. • `spread`: Implies that threads are bound and distributed as far apart as possible, based on the unit of `OMP_PLACES`. `spread` places OpenMP threads as far away from each other as possible, which is good for resource utilization (FP units, memory bandwidth). • `master`: The OpenMP worker threads are in the same place partition as the master thread. **Note:** `spread` or `close` placement of OpenMP threads only matters when you are under-populating a node.
`OMP_PLACES={threads|cores|sockets}|{<lower-bound>:<length>:<stride>}`	Specifies the thread placement (threads, cores, or sockets), and can also be used for explicit thread placement. For example, where `OMP_PLACES={0:2:2},{4:2:2},{8:2:2}` means: • T0: 0,2 • T1: 4,6 • T2: 8,10
`OMP_DISPLAY_ENV={true|false|verbose}`	Controls what OpenMP environment information displays on application on startup. `OMP_DISPLAY_ENV` can be set to: • `true`: On startup, your application displays the version of OpenMP along with the value for all of the OpenMP Internal Control Variables (ICVs) which are affected by environment variables in this topic, and other factors. **Note:** There might be a discrepancy between the value of your environment variables and the ICVs reported at runtime because ICVs can be controlled in other ways. • `false`: No output is produced. • `verbose`: The values of any implementation-specific variables are displayed in addition to the standard OpenMP ICVs.

If you are using Arm® Compiler for Linux, Arm also recommends that you set `KMP_AFFINITY=verbose` to help you to understand the thread placement on your system and the impact on runtime, so that you can choose the best configuration for your application.

Note

`KMP_AFFINITY` only works for LLVM-based compilers, it does not work for GNU compilers. If you are using a GNU compiler, you can use `OMP_DISPLAY_ENV=VERBOSE` to list all the OpenMP environment variables, however it does not give the core mappings.

OpenMP thread placement basics

To avoid multithreading performance problems when using Arm Compiler for Linux, it is important that you have the appropriate environment set up.

Set the number of OpenMP threads

To set the number of threads to use in your program, set the environment variable `OMP_NUM_THREADS`. `OMP_NUM_THREADS` sets the number of threads used in OpenMP parallel regions defined in your own code, and within Arm Performance Libraries. If you set `OMP_NUM_THREADS` to a single value, your program uses a single level of parallelism. In this case, nested parallelism is disabled.

Note

The information about setting `OMP_NUM_THREADS` applies to both compilers supported by Arm Performance Libraries in the 23.04 release: Arm C/C++/Fortran Compiler 23.04 and GCC 12.2.

For example, consider the following code, `threading.c`, which defines a nested parallel region:

```
#include <stdio.h>
#include <omp.h>

int main() {
      #pragma omp parallel
      {
               printf("outer: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
               #pragma omp parallel
               {
                  printf("inner: omp_get_thread_num = %d omp_get_level = %d\n",
 omp_get_thread_num(), omp_get_level());
               }
      }
}
```

Compiling and running the code gives the following output:

- Arm Compiler for Linux, building the executable `a1.out`:

```
armclang -o a1.out -fopenmp threading.c
OMP_NUM_THREADS=2 ./a1.out

outer: omp_get_thread_num = 0 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
```

- GCC, building the executable `g1.out`:

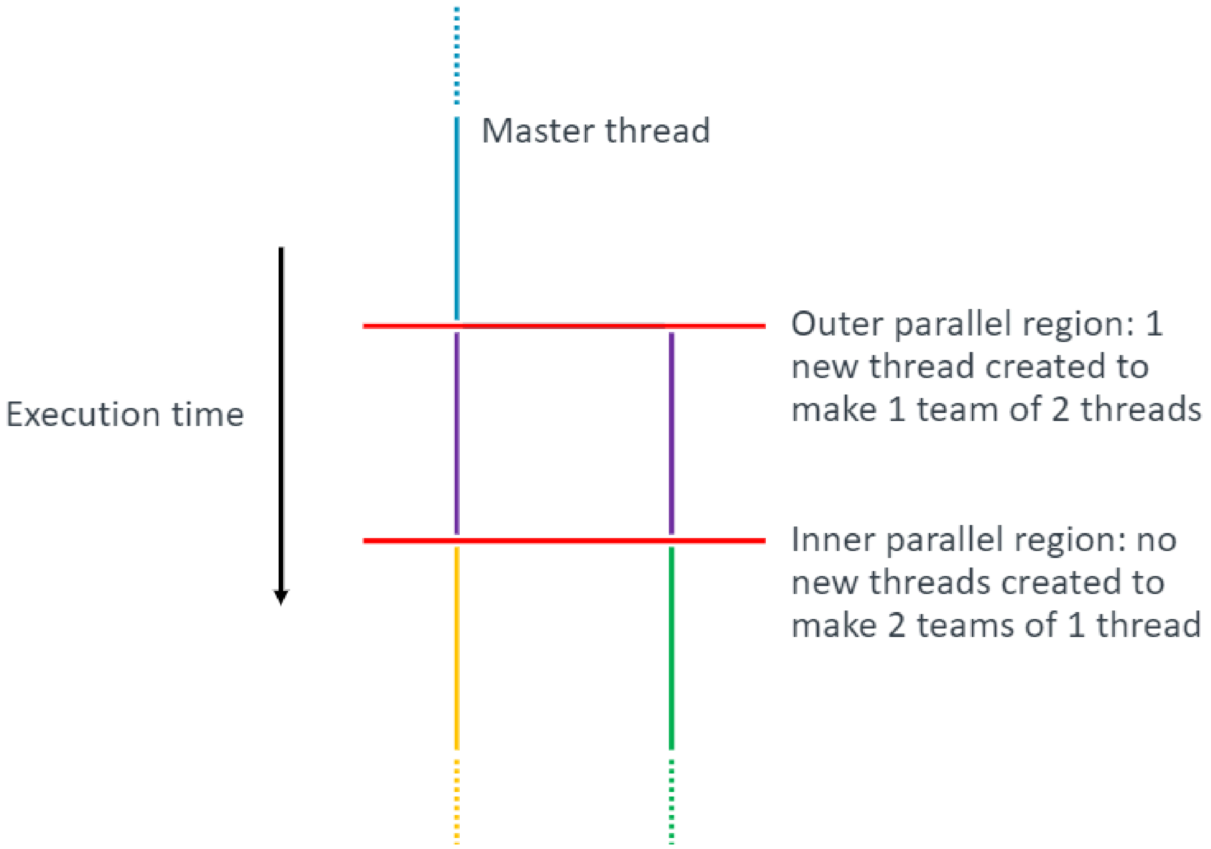
```
gcc -o g1.out -fopenmp threading.c
OMP_NUM_THREADS=2 ./g1.out

outer: omp_get_thread_num = 0 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
outer: omp_get_thread_num = 1 omp_get_level = 1
```

```
inner: omp_get_thread_num = 0 omp_get_level = 2
```

The program above reports the thread number and level of parallel nesting. Executables that are built with either GCC or Arm Compiler for Linux show the same behavior when `OMP_NUM_THREADS` is set to a single value (and all other settings use default values).

The example given earlier sets `OMP_NUM_THREADS=2` and the output shows that two threads are used for the outer parallel region. The nested parallel regions do not create any new threads:

Figure 3-2: No nested parallelism diagram.


Master thread
Execution time
Outer parallel region: 1 new thread created to make 1 team of 2 threads
Inner parallel region: no new threads created to make 2 teams of 1 thread

Note

The actual number of threads used during execution of your program might differ from the value specified in `OMP_NUM_THREADS`. The number of threads can differ if the number of threads is set explicitly in the code using the OpenMP API, or if a system-defined limit is encountered.

`OMP_NUM_THREADS` can also be set to a comma-separated list of values. Where a list of values is passed to `OMP_NUM_THREADS`, the values denote the number of threads to use at each level of nesting, starting from the outermost parallel region.

- Arm Compiler for Linux:

```
OMP_NUM_THREADS=2,2 ./a1.out
```

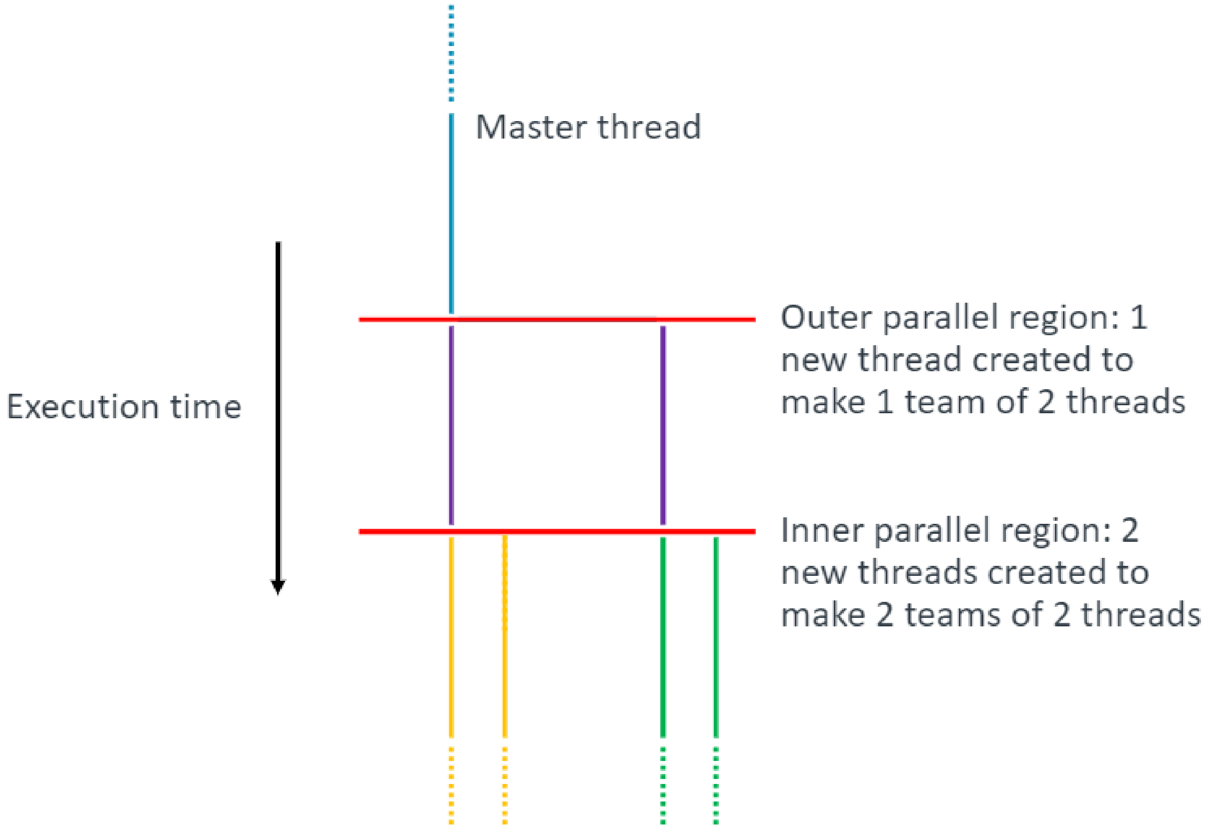
```
outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
```

- GCC:

```
OMP_NUM_THREADS=2,2 ./g1.out

outer: omp_get_thread_num = 0 omp_get_level = 1
outer: omp_get_thread_num = 1 omp_get_level = 1
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 0 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
inner: omp_get_thread_num = 1 omp_get_level = 2
```

The examples above specify that the two parallel regions in the code can each use two threads. With both compilers, the executables create a new thread in each of the two inner parallel regions, enabling nested parallelism.

Figure 3-3: Nested parallelism diagram.


Master thread
Execution time
Outer parallel region: 1 new thread created to make 1 team of 2 threads
Inner parallel region: 2 new threads created to make 2 teams of 2 threads

Note

- Defaulting to nested parallelism was a change of behavior for executables linked to the OpenMP runtime in Arm Compiler for Linux version 20.0 and GCC version 11.0. In earlier compiler versions, the default was for nested parallelism to be disabled.
- The OMP_NESTED setting is being deprecated for OpenMP 5.0. Instead, we recommend that you use OMP_MAX_ACTIVE_LEVELS to control the depth of nesting.

Nested parallelism in Arm Performance Libraries is handled in the same way as shown in these examples; if an Arm Performance Libraries routine is called from a parallel region in your code, then the routine spawns threads in the same way as shown for the nested parallel region in the examples above.

Control the placement of threads

The value of the environment variable `OMP_PROC_BIND` affects how threads are assigned to cores on your system (also known as thread affinity). If `OMP_PROC_BIND=false` or is unset, then threads are unpinned. Unpinned threads might be migrated between cores in the system during execution, and thread migration will most likely degrade performance significantly.

Arm recommends setting `OMP_PROC_BIND` to either `true`, `close` or `spread`, as required:

- If `close` is set, then the OpenMP threads are pinned to cores close to the parent thread. `OMP_PROC_BIND=close` is useful where threads in a team are working on locally shared data. For example, if threads are pinned to neighboring cores there might be a performance benefit from the data being stored in a shared level of cache.
- If `spread` is set, then the OpenMP threads are pinned to cores that are distant from the parent thread. OMP_PROC_BIND=spread is useful to avoid contention on hardware resources. For example, if threads are working on large amounts of private data then there might be an advantage to using `spread`. Using `spread` can reduce contention on a shared level of cache or memory bandwidth.
- If `true` is set, thread migration is avoided and no affinity policy is specified.
- If `master` is set, all OpenMP threads in a team are pinned to the same core as the master thread.

To set the affinity policy separately for each level of nested parallelism, set `OMP_PROC_BIND` to a comma-separated list of the values described above.

Note

The values assigned to OpenMP environment variables are case insensitive.

The statements above describe how OpenMP threads are pinned to cores in the system. However, the OpenMP specification uses the term "place" to denote a hardware resource for which threads can have affinity. The environment variable `OMP_PLACES` allows you to define what is meant by a "place" in the system.

`OMP_PLACES` can be set to one of three pre-defined values: `threads`, `cores` or `sockets`. Setting `OMP_PLACES=threads` assigns OpenMP threads to hardware threads in the system. On a system where a single core supports multiple hardware threads (for example, Marvell ThunderX2 systems with SMT>1), assigning OpenMP threads to hardware threads allows for the co-location of several threads in a single core.

If the value is set to `cores`, each OpenMP thread is assigned to a different core in the system, which might support more than one hardware thread.

If the value is set to `sockets`, each OpenMP thread is assigned to a single socket in the system, which contains multiple cores. Where `sockets` is set, the OpenMP threads might migrate in the assigned socket.

To more finely control the placement of OpenMP threads in your system, set `OMP_PLACES` to a list of numbers that indicate the IDs of hardware places in your system (typically hardware threads). There is a considerable amount of flexibility availability using `OMP_PLACES`, including the ability to exclude places from thread placement. If you are interested in this level of control, refer to the OpenMP specification and experiment on your system.

Simultaneous Multithreading (SMT)

With the high core counts on server scale AArch64 CPUs, you might experience an unexpected performance degradation of OpenMP codes because of the kernel swapping threads.

Arm recommends that you set the key environment variables (see previous section) and make an informed choice based on the parallel programming model and system configuration (for example, Simultaneous Multithreading (SMT) `SMT={1,2,4}`) being used for your code.

Note

For `SMT=1`, `threads` is equivalent to `cores`, but where SMT is > 1 you must decide if you want to pin to the local core (`threads`) or the physical core (`core`).

`SMT=1`

Default settings (unset): `OMP_NUM_THREADS=<number of logical cores available on the system>`, `OMP_PROC_BIND=false`, and `OMP_PLACES=cores`.

For example, in a 64-core system the number of logical cores (given by the product of the number of physical cores and the number of SMT threads) is 64. Therefore, `OMP_NUM_THREADS=64`.

Job launches with all available OpenMP threads, each 'bound' to the full processor set (in other words, free to move between any physical core).

- `OMP_PROC_BIND=close` and `OMP_PLACES=cores`

Job launches with all available OpenMP threads, each 'bound' to one core, filling sequentially: socket 0, then socket 1. The OpenMP threads are pinned to one physical core.

- `OMP_PROC_BIND=close` and `OMP_PLACES=sockets`

  Job launches with all available OpenMP threads, each 'bound' to one socket, filling sequentially: socket 0, then socket 1. The OpenMP threads are free to swap within that socket, before being evenly distributed between the sockets.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=cores`

  Job launches with all available OpenMP threads, each 'bound' to one core and spreading out as far away from each other as possible.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=sockets`

  Job launches with all available OpenMP threads, each 'bound' to one socket, filling sequentially: socket 0, then socket 1. The OpenMP threads spread out as far away from each other as possible within that socket, before being evenly distributed between the sockets.

`SMT={2|4}`

Default settings (unset): `OMP_NUM_THREADS=<number of logical cores available on the system>`, `OMP_PROC_BIND=false`, and `OMP_PLACES=cores`.

For example, in a 64-core system the number of logical cores is 256 cores. Therefore, `OMP_NUM_THREADS=256`.

Job launches with all available OpenMP threads, each 'bound' to the full processor set (in other words, free to move between either thread on any physical core).

- `OMP_PROC_BIND=close` and `OMP_PLACES=cores`

  Job launches with all available OpenMP threads, two (`SMT=2`) or four (`SMT=4`) per core, OpenMP threads pinned to these sets. Each OpenMP thread can migrate between all the hardware threads on a physical core. Each core belongs to a single OpenMP thread. Cores are filled sequentially: socket 0, then socket 1. All cores on socket are filled before moving onto the next socket.

- `OMP_PROC_BIND=close` and `OMP_PLACES=threads`

  Pins each process to one slot on one core. Available slots are filled, as with `OMP_PLACES=cores`.

- `OMP_PROC_BIND=close` and `OMP_PLACES=sockets`

  Pins each process to any thread on a socket, but fills all the cores (not all the slots) on one socket, before moving onto the next. In other words, each OpenMP thread gets a single hardware thread and are packed in. Behaves the same as `OMP_PROC_BIND=spread OMP_PLACES=sockets`.

- `OMP_PROC_BIND=spread` and `OMP_PLACES=cores`

Job launches with all available OpenMP threads, each bound to a core and free to migrate between the two or four slots. OpenMP threads are spread between available sockets (in other words, one thread per core until physical cores are all in use).

- `OMP_PROC_BIND=spread` and `OMP_PLACES=sockets`

  Pins each process to any thread on a socket, but fills all the cores (not all the slots) on one socket, before moving onto the next.

Example: Investigating an OpenMP where `SMT=1`

Note

This example uses a 64-core node and only 8 cores for illustration purposes.

1. Run your OpenMP application, for example `example-01`, and specify the number of OpenMP threads to run it on. For this example, 8 are specified:

```
KMP_AFFINITY=verbose OMP_NUM_THREADS=8  ./example-01
```

This gives you an output similar to:

```
...
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected:
  {0,1,2,...,62,63}
OMP: Info #156: KMP_AFFINITY: 64 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 32 cores/pkg x 1 threads/core
  (64 total cores)
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24047 thread 0 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24048 thread 1 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24049 thread 2 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24050 thread 3 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24051 thread 4 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24052 thread 5 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24053 thread 6 bound to OS proc set
 {0,1,2,...,62,63}
OMP: Info #248: KMP_AFFINITY: pid 24047 tid 24054 thread 7 bound to OS proc set
 {0,1,2,...,62,63}
...
```

Where:

- All 64 (0 to 63) physical cores are shown, split over two sockets of 32 cores, and each physical core showing one thread per core.
- There are no bindings or mappings between logical cores and physical cores.

- The Process ID (PID) shows the full logical core set being available to each of the eight OpenMP threads. The full set of logical cores being available is because `OMP_PROC_BIND` is unset and defaults to `OMP_PROC_BIND=false`.

2. To control how the threads are distributed, set `OMP_PROC_BIND=true`:

```
KMP_AFFINITY=verbose OMP_PROC_BIND=true OMP_NUM_THREADS=8  ./example-01
```

This output is now similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24055 tid 24055 thread 0 bound to OS proc set
 {0}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24056 thread 1 bound to OS proc set
 {8}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24057 thread 2 bound to OS proc set
 {16}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24058 thread 3 bound to OS proc set
 {24}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24059 thread 4 bound to OS proc set
 {32}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24060 thread 5 bound to OS proc set
 {40}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24061 thread 6 bound to OS proc set
 {48}
OMP: Info #248: OMP_PROC_BIND: pid 24055 tid 24062 thread 7 bound to OS proc set
 {56}
...
```

Where:

- The same 64 logical cores are available.
- Each logical core is now 'bound' (mapped) to a physical core because the `OMP_PROC_BIND` is set: logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map logical cores 32-63 to physical cores 0-31 again.
- The Process ID (PID) shows the eight OpenMP threads being evenly distributed over the 64 logical cores: {0}, {8}, .., {56}. This spread distribution is because the combination of `OMP_PROC_BIND=true` and an unset `OMP_PLACES` results in `OMP_PROC_BIND=spread` effectively being set.

3. To bring the threads closer together, use `OMP_PROC_BIND=close`:

```
KMP_AFFINITY=verbose OMP_PROC_BIND=close OMP_NUM_THREADS=8  ./example-01
```

The output is now similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
OMP: Info #156: KMP_AFFINITY: 64 available OS procs
```

```
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 32 cores/pkg x 1 threads/core
   (64 total cores)
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24063 tid 24063 thread 0 bound to OS proc set
 {0}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24064 thread 1 bound to OS proc set
 {1}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24065 thread 2 bound to OS proc set
 {2}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24066 thread 3 bound to OS proc set
 {3}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24067 thread 4 bound to OS proc set
 {4}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24068 thread 5 bound to OS proc set
 {5}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24069 thread 6 bound to OS proc set
 {6}
OMP: Info #248: OMP_PROC_BIND: pid 24063 tid 24070 thread 7 bound to OS proc set
 {7}
...
```

Where:

- The same 64 logical cores are available.
- Each logical core is still mapped to a physical core because the `OMP_PROC_BIND` is set: logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map logical cores 32-63 to physical cores 0-31 again.
- The Process ID (PID) shows the eight OpenMP threads being closely distributed over the first 8 of the 64 logical cores: {0}, {1}, .., {8}. This close distribution is because `OMP_PROC_BIND` is now set to `OMP_PROC_BIND=close`.

4. If your application has specific functions being set to specific OpenMP threads, it can be useful to weight the cores according to the computational intensity of these functions.

Note

This will vary from application to application.

To precisely distribute the OpenMP threads to specific logical cores, you can explicitly set these for `OMP_PLACES`. For example, set `OMP_PLACES={0:2}:8:4` to request that each OpenMP thread is given two logical cores, with eight multiples of two logical cores ({0:2}), spaced four logical cores apart:

```
KMP_AFFINITY=verbose OMP_PLACES={0:2}:8:4 OMP_NUM_THREADS=8  ./example-01
```

The output is similar to:

```
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,...,62,63}
...
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3
...
OMP: Info #171: KMP_AFFINITY: OS proc 31 maps to package 0 core 31
OMP: Info #171: KMP_AFFINITY: OS proc 32 maps to package 1 core 0
...
OMP: Info #248: KMP_AFFINITY: pid 24071 tid 24071 thread 0 bound to OS proc set
 {0,1}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24072 thread 1 bound to OS proc set
 {4,5}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24073 thread 2 bound to OS proc set
 {8,9}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24074 thread 3 bound to OS proc set
 {12,13}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24075 thread 4 bound to OS proc set
 {16,17}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24076 thread 5 bound to OS proc set
 {20,21}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24077 thread 6 bound to OS proc set
 {24,25}
OMP: Info #248: OMP_PROC_BIND: pid 24071 tid 24078 thread 7 bound to OS proc set
 {28,29}
...
```

Where:

- The same 64 logical cores are available.
- Each logical core is still mapped to a physical core because the `OMP_PROC_PLACES` is set: logical cores 0-31 are mapped to physical cores 0-31, before the system cycles back to map logical cores 32-63 to physical cores 0-31 again.
- The Process ID (PID) shows the eight OpenMP threads being explicitly places onto two logical cores each, at a distance of 4 logical cores apart: {0,1}, {4,5}, .., {28,29}. This explicit placement is because `OMP_PLACES` is set to `OMP_PLACES={0:2}:8:4`.

Affinity in a hybrid environment: Open MPI

Hybrid applications are applications that employ multiple parallel programming models, for example MPI and OpenMP. Running hybrid applications introduces a second layer of complexity. By default, MPI runtimes provide options to control thread and task placement, and might set bindings for MPI processes.

Note

This topic uses Open MPI as an example MPI implementation to discuss the concepts being described. Other MPI implementations are available, such as MPICH and MVAPICH2. For recipes to port your MPI implementation to Arm, see the Porting and Tuning web page.

Some common Open MPI runtime options include:

`--bind-to [hwthread\|core\|socket]`

Controls what unit of hardware to bind threads to.

`--map-by [hwthread\|core\|slot][:PE=x]`

Controls the distribution of resources to MPI ranks. `x` in `PE=x` is how many processing elements to give to each rank. For example, to give two tasks per socket using `--map-by`, use: `--map-by ppr:2:socket`.

`--report-bindings`

Report the MPI ranks bindings for launched processes. `report-bindings` is useful for obtaining information on how tasks and OpenMP threads are mapped and bound at run-time.

`--display-map`

Display a table showing the mapped location of each process ahead of launch.

Useful for obtaining information on how tasks and OpenMP threads that are mapped and bound at run-time.

`-use-hwthread-cpus`

Treat SMT hardware threads as distinct cores.

Example: Investigating a hybrid OpenMPI and OpenMP application where `SMT=1`

Note

This example uses a 64-core node and only launches 4 Open MPI processes per node for illustration purposes.

1. Run your hybrid application, for example `example-02`, and specify the number of OpenMP threads (in this case four) and Open MPI nodes (in this case four) to run it on. This example sets the Open MPI options `--map-by slot:PE=4` to distribute the MPI ranks over slots, and use 4 processing elements. `-bind-to core` binds the MPI processes to logical cores:

   ```
   KMP_AFFINITY=verbose  OMP_NUM_THREADS=4 mpirun -np 4 --map-by slot:PE=4 --bind-to
    core --report-bindings  ./example-02
   ```

   This give you the following extracts from the output:

   ```
   MCW rank 0 bound to socket 0[core 0[hwt 0]],
     socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]]:
     [B/B/B/B/./././././././././././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././.]
   MCW rank 1 bound to socket 0[core 4[hwt 0]],
     socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]], socket 0[core 7[hwt 0]]:
     [././././B/B/B/B/././././././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././.]
   MCW rank 2 bound to socket 0[core 8[hwt 0]],
     socket 0[core 9[hwt 0]], socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]]:
     [././././././././B/B/B/B/./././././././././././././././././././.]
   [./././././././././././././././././././././././././././././././.]
   MCW rank 3 bound to socket 0[core 12[hwt 0]],
     socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
     [./././././././././././././B/B/B/B/./././././././././././././././.]
   [./././././././././././././././././././././././././././././././.]
   ```

```
...
OMP: Info #179: KMP_AFFINITY: 1 packages x 4 cores/pkg x 1 threads/core (4 total
 cores)
OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24635 thread 0 bound to OS proc set
 {0,1,2,3}
OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24684 thread 1 bound to OS proc set
 {0,1,2,3}
OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24687 thread 2 bound to OS proc set
 {0,1,2,3}
OMP: Info #248: KMP_AFFINITY: pid 24635 tid 24690 thread 3 bound to OS proc set
 {0,1,2,3}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24636 thread 0 bound to OS proc set
 {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24689 thread 1 bound to OS proc set
 {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24691 thread 2 bound to OS proc set
 {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24636 tid 24693 thread 3 bound to OS proc set
 {4,5,6,7}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24637 thread 0 bound to OS proc set
 {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24692 thread 1 bound to OS proc set
 {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24694 thread 2 bound to OS proc set
 {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24637 tid 24695 thread 3 bound to OS proc set
 {8,9,10,11}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24639 thread 0 bound to OS proc set
 {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24685 thread 1 bound to OS proc set
 {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24686 thread 2 bound to OS proc set
 {12,13,14,15}
OMP: Info #248: KMP_AFFINITY: pid 24639 tid 24688 thread 3 bound to OS proc set
 {12,13,14,15}
...
```

Where:

- Each MPI rank is bound to a set of 4 logical cores, sequentially filling the first 16 of the available 64 logical cores.
- The Process ID (PID) shows each MPI process (0-3) receiving an OpenMP thread (four of each process) each with the OpenMP thread able to be placed on any of the next 4 logical cores in the core set: ranks 0 get {0,1,2,3},..., {12,13,14,15}, ..., ranks 3 get {0,1,2,3},.., {12,13,14,15}.
- The binding of the OpenMP threads cycles over 0-15 for each processing element is because `--bind-to core` is set.

2. To introduce more control into the location of the OpenMP threads, bind the OpenMP threads to cores. Set `OMP_PROC_BIND=true`:

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=true OMP_NUM_THREADS=4 mpirun -np 4 --map-by
 slot:PE=4 --bind-to core --report-bindings  ./example-02
```

This give you the following extracts from the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]]:
  [B/B/B/B/./././././././././././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 4[hwt 0]],
  socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]], socket 0[core 7[hwt 0]]:
  [././././B/B/B/B/././././././././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 0[core 8[hwt 0]],
  socket 0[core 9[hwt 0]], socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]]:
  [././././././././B/B/B/B/././././././././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 3 bound to socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [././././././././././././B/B/B/B/./././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
...
OMP: Info #248: KMP_AFFINITY: pid 24780 tid 24780 thread 0 bound to OS proc set
 {0}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24846 thread 1 bound to OS proc set
 {1}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24850 thread 2 bound to OS proc set
 {2}
OMP: Info #248: OMP_PROC_BIND: pid 24780 tid 24853 thread 3 bound to OS proc set
 {3}
OMP: Info #248: KMP_AFFINITY: pid 24781 tid 24781 thread 0 bound to OS proc set
 {4}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24847 thread 1 bound to OS proc set
 {5}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24849 thread 2 bound to OS proc set
 {6}
OMP: Info #248: OMP_PROC_BIND: pid 24781 tid 24856 thread 3 bound to OS proc set
 {7}
OMP: Info #248: KMP_AFFINITY: pid 24782 tid 24782 thread 0 bound to OS proc set
 {8}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24845 thread 1 bound to OS proc set
 {9}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24851 thread 2 bound to OS proc set
 {10}
OMP: Info #248: OMP_PROC_BIND: pid 24782 tid 24855 thread 3 bound to OS proc set
 {11}
OMP: Info #248: KMP_AFFINITY: pid 24783 tid 24783 thread 0 bound to OS proc set
 {12}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24848 thread 1 bound to OS proc set
 {13}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24852 thread 2 bound to OS proc set
 {14}
OMP: Info #248: OMP_PROC_BIND: pid 24783 tid 24854 thread 3 bound to OS proc set
 {15}
...
```

Where:

- Each MPI rank is still bound to a set of 4 logical cores, sequentially filling the first 16 of the available 64 logical cores.
- The Process ID (PID) shows each MPI process (0-3) receiving an OpenMP thread (four of each process) each with the OpenMP thread bound to the most spread available logical core: ranks 0 get {0},{4},{8},{12},..., ranks 3 get {3},{7},{11},{15}. The core-bound distribution of OpenMP threads is because `OMP_PROC_BIND=true` is set.

3. Similar to the OpenMP example, the distribution of th OpenMP threads can be forced to be placed closer by setting `OMP_PROC_BIND=close`.

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=close OMP_NUM_THREADS=4 mpirun -np 4 --map-by
 slot:PE=16 --bind-to core --report-bindings  ./example-02
```

Giving you the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]],
  socket 0[core 4[hwt 0]], socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]],
  socket 0[core 7[hwt 0]], socket 0[core 8[hwt 0]], socket 0[core 9[hwt 0]],
  socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]], socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 16[hwt 0]],
  socket 0[core 17[hwt 0]], socket 0[core 18[hwt 0]], socket 0[core 19[hwt 0]],
  socket 0[core 20[hwt 0]], socket 0[core 21[hwt 0]], socket 0[core 22[hwt 0]],
  socket 0[core 23[hwt 0]], socket 0[core 24[hwt 0]], socket 0[core 25[hwt 0]],
  socket 0[core 26[hwt 0]], socket 0[core 27[hwt 0]], socket 0[core 28[hwt 0]],
  socket 0[core 29[hwt 0]], socket 0[core 30[hwt 0]], socket 0[core 31[hwt 0]]:
  [./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
[./././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 1[core 32[hwt 0]],
  socket 1[core 33[hwt 0]], socket 1[core 34[hwt 0]], socket 1[core 35[hwt 0]],
  socket 1[core 36[hwt 0]], socket 1[core 37[hwt 0]], socket 1[core 38[hwt 0]],
  socket 1[core 39[hwt 0]], socket 1[core 40[hwt 0]], socket 1[core 41[hwt 0]],
  socket 1[core 42[hwt 0]], socket 1[core 43[hwt 0]], socket 1[core 44[hwt 0]],
  socket 1[core 45[hwt 0]], socket 1[core 46[hwt 0]], socket 1[core 47[hwt 0]]:
  [./././././././././././././././././././././././././././././././.][B/B/B/B/B/B/
B/B/B/B/B/B/B/B/B/B/./././././././././././././././.]
MCW rank 3 bound to socket 1[core 48[hwt 0]],
 socket 1[core 49[hwt 0]], socket 1[core 50[hwt 0]], socket 1[core 51[hwt 0]],
 socket 1[core 52[hwt 0]], socket 1[core 53[hwt 0]], socket 1[core 54[hwt 0]],
 socket 1[core 55[hwt 0]], socket 1[core 56[hwt 0]], socket 1[core 57[hwt 0]],
 socket 1[core 58[hwt 0]], socket 1[core 59[hwt 0]], socket 1[core 60[hwt 0]],
 socket 1[core 61[hwt 0]], socket 1[core 62[hwt 0]], socket 1[core 63[hwt 0]]:
 [./././././././././././././././././././././././././././././././.]
[./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
...
OMP: Info #248: KMP_AFFINITY: pid 24878 tid 24878 thread 0 bound to OS proc set
 {0}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24927 thread 1 bound to OS proc set
 {1}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24927 thread 2 bound to OS proc set
 {2}
OMP: Info #248: OMP_PROC_BIND: pid 24878 tid 24935 thread 3 bound to OS proc set
 {3}
OMP: Info #248: KMP_AFFINITY: pid 24879 tid 24879 thread 0 bound to OS proc set
 {16}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24936 thread 1 bound to OS proc set
 {17}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24937 thread 2 bound to OS proc set
 {18}
OMP: Info #248: OMP_PROC_BIND: pid 24879 tid 24938 thread 3 bound to OS proc set
 {19}
OMP: Info #248: KMP_AFFINITY: pid 24881 tid 24881 thread 0 bound to OS proc set
 {32}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24928 thread 1 bound to OS proc set
 {33}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24930 thread 2 bound to OS proc set
 {34}
OMP: Info #248: OMP_PROC_BIND: pid 24881 tid 24931 thread 3 bound to OS proc set
 {35}
OMP: Info #248: KMP_AFFINITY: pid 24882 tid 24882 thread 0 bound to OS proc set
 {48}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24929 thread 1 bound to OS proc set
 {49}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24932 thread 2 bound to OS proc set
 {50}
OMP: Info #248: OMP_PROC_BIND: pid 24882 tid 24934 thread 3 bound to OS proc set
 {51}
```

```
...
```

Where:

- Each MPI rank is uses 16 processing elements bound to a set of 16 logical cores, sequentially filling the available 64 logical cores.
- The Process ID (PID) shows each MPI process (0-3) having four OpenMP threads (four of each process) where the OpenMP threads are bound to the closest available logical core: ranks 0 get {0},{16},{32},{48},..., ranks 3 get {3},{19},{35},{51}. The close distribution of OpenMP threads is because `OMP_PROC_BIND=close` is set. The closest logical cores are 16 apart because of the 16 processing elements.

4. Setting `OMP_PROC_BIND=spread` shows how the threads can be distributed as far away from each other:

```
KMP_AFFINITY=verbose  OMP_PROC_BIND=spread OMP_NUM_THREADS=4 mpirun -np 4 --map-
by slot:PE=16 --bind-to core --report-bindings  ./example-02
```

Giving you the output:

```
MCW rank 0 bound to socket 0[core 0[hwt 0]],
  socket 0[core 1[hwt 0]], socket 0[core 2[hwt 0]], socket 0[core 3[hwt 0]],
  socket 0[core 4[hwt 0]], socket 0[core 5[hwt 0]], socket 0[core 6[hwt 0]],
  socket 0[core 7[hwt 0]], socket 0[core 8[hwt 0]], socket 0[core 9[hwt 0]],
  socket 0[core 10[hwt 0]], socket 0[core 11[hwt 0]], socket 0[core 12[hwt 0]],
  socket 0[core 13[hwt 0]], socket 0[core 14[hwt 0]], socket 0[core 15[hwt 0]]:
  [B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/./././././././././././././././.]
[./././././././././././././././././././././././././././././././.]
MCW rank 1 bound to socket 0[core 16[hwt 0]],
  socket 0[core 17[hwt 0]], socket 0[core 18[hwt 0]], socket 0[core 19[hwt 0]],
  socket 0[core 20[hwt 0]], socket 0[core 21[hwt 0]], socket 0[core 22[hwt 0]],
  socket 0[core 23[hwt 0]], socket 0[core 24[hwt 0]], socket 0[core 25[hwt 0]],
  socket 0[core 26[hwt 0]], socket 0[core 27[hwt 0]], socket 0[core 28[hwt 0]],
  socket 0[core 29[hwt 0]], socket 0[core 30[hwt 0]], socket 0[core 31[hwt 0]]:
  [./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
[./././././././././././././././././././././././././././././././.]
MCW rank 2 bound to socket 1[core 32[hwt 0]],
  socket 1[core 33[hwt 0]], socket 1[core 34[hwt 0]], socket 1[core 35[hwt 0]],
  socket 1[core 36[hwt 0]], socket 1[core 37[hwt 0]], socket 1[core 38[hwt 0]],
  socket 1[core 39[hwt 0]], socket 1[core 40[hwt 0]], socket 1[core 41[hwt 0]],
  socket 1[core 42[hwt 0]], socket 1[core 43[hwt 0]], socket 1[core 44[hwt 0]],
  socket 1[core 45[hwt 0]], socket 1[core 46[hwt 0]], socket 1[core 47[hwt 0]]:
  [./././././././././././././././././././././././././././././././.][B/B/B/B/B/B/
B/B/B/B/B/B/B/B/B/B/./././././././././././././././.]
MCW rank 3 bound to socket 1[core 48[hwt 0]],
  socket 1[core 49[hwt 0]], socket 1[core 50[hwt 0]], socket 1[core 51[hwt 0]],
  socket 1[core 52[hwt 0]], socket 1[core 53[hwt 0]], socket 1[core 54[hwt 0]],
  socket 1[core 55[hwt 0]], socket 1[core 56[hwt 0]], socket 1[core 57[hwt 0]],
  socket 1[core 58[hwt 0]], socket 1[core 59[hwt 0]], socket 1[core 60[hwt 0]],
  socket 1[core 61[hwt 0]], socket 1[core 62[hwt 0]], socket 1[core 63[hwt 0]]:
  [./././././././././././././././././././././././././././././././.]
[./././././././././././././././././B/B/B/B/B/B/B/B/B/B/B/B/B/B/B/B]
...
OMP: Info #248: KMP_AFFINITY: pid 24977 tid 24977 thread 0 bound to OS proc set
 {0}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25028 thread 1 bound to OS proc set
 {4}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25031 thread 2 bound to OS proc set
 {8}
OMP: Info #248: OMP_PROC_BIND: pid 24977 tid 25036 thread 3 bound to OS proc set
 {12}
```

```
OMP: Info #248: KMP_AFFINITY: pid 24978 tid 24978 thread 0 bound to OS proc set
 {16}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25027 thread 1 bound to OS proc set
 {20}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25030 thread 2 bound to OS proc set
 {24}
OMP: Info #248: OMP_PROC_BIND: pid 24978 tid 25035 thread 3 bound to OS proc set
 {28}
OMP: Info #248: KMP_AFFINITY: pid 24979 tid 24979 thread 0 bound to OS proc set
 {32}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25025 thread 1 bound to OS proc set
 {36}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25029 thread 2 bound to OS proc set
 {40}
OMP: Info #248: OMP_PROC_BIND: pid 24979 tid 25033 thread 3 bound to OS proc set
 {44}
OMP: Info #248: KMP_AFFINITY: pid 24980 tid 24980 thread 0 bound to OS proc set
 {48}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25026 thread 1 bound to OS proc set
 {52}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25032 thread 2 bound to OS proc set
 {56}
OMP: Info #248: OMP_PROC_BIND: pid 24980 tid 25034 thread 3 bound to OS proc set
 {60}
...
```

Where:

- Each MPI rank is uses 16 processing elements bound to a set of 16 logical cores, sequentially filling the available 64 logical cores.
- The Process ID (PID) shows each MPI process (0-3) having four OpenMP threads (four of each process) where the OpenMP threads are bound to the furthest available logical core: ranks 0 get {0},{16},{32},{48},..., ranks 3 get {12},{28},{44},{60}. The spread distribution of OpenMP threads is because `OMP_PROC_BIND` is set to `spread`. The furthermost logical cores are 4 apart.

lstopo

`lstopo` can provide a simple representation of where the available hardware threads are located. In other words, it is convenient for deciphering the numbering scheme.

To install and run `lstopo`:

1. Install hwloc and hwloc-gui. Use a suitable package installer for your OS (you might need root permissions).

   For example, on a CentOS or RedHat system using `yum`:

   ```
   yum install hwloc hwloc-gui -y
   ```

2. Run `lstopo`:

   - To run using the GUI, use:

     ```
     lstopo -p
     ```

- To generate a pdf, use:

```
lstopo -p --output-format pdf > topology.pdf
```

- To generate a text output, use:

```
lstopo -p --output-format console
```

Tips for application porting and optimization

- If your application uses OpenMP, Arm recommends that you set key OpenMP environment variables like `OMP_NUM_THREADS` and `OMP_PROC_BIND`.
- To identify the optimum choices, run your application on a number of core counts and using various values for `OMP_PROC_BIND` and `OMP_PLACES`.
- To profile the application and identify any significant OpenMP overheads and bottlenecks, use Arm MAP.
- Set your key environment variables. Make an informed choice based on the parallel programming model and system configuration (for example `SMT={1,2,4}`, or using numactl) being used for your code.
- To understand the thread placement on your system and the impact on run-time, so that you can choose the best configuration for your application, set `KMP_AFFINITY=verbose` and use `lstopo`.
- Consider the binding options available in the MPI distribution you are using.

3.5 Emulate your application with Arm Instruction Emulator

This chapter briefly describes Arm® Instruction Emulator (ArmIE), Arm's SVE instruction emulator tool.

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 on hardware that is not SVE-enabled.

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.

Note

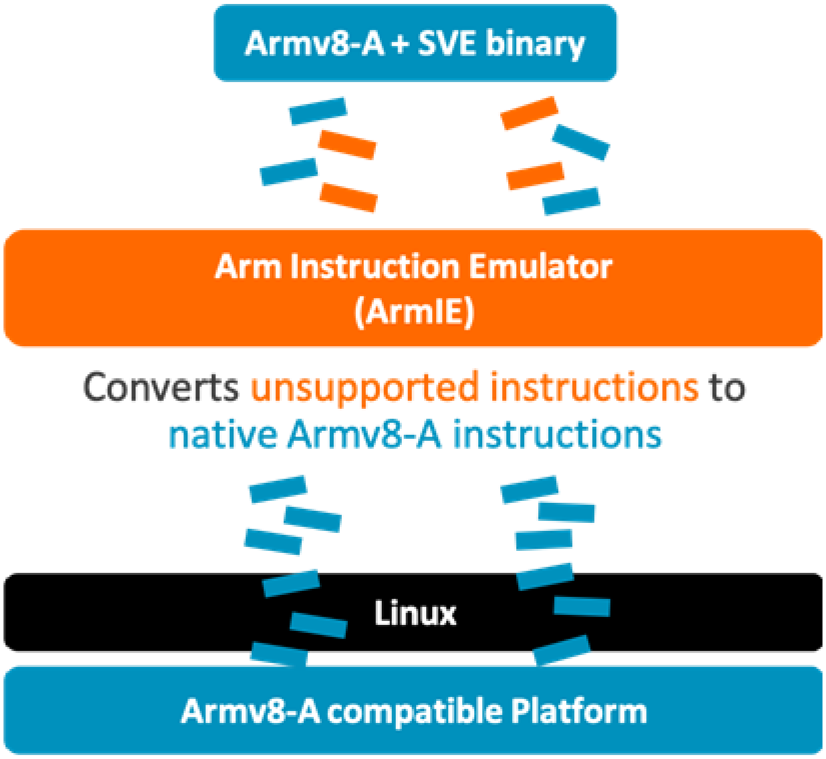
The latest information about Arm Instruction Emulator is available on the Arm Instruction Emulator developer webpage and in the Arm Instruction Emulator Developer and Reference Guide.

3.5.1 Introduction to Arm Instruction Emulator (ArmIE)

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

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.

Figure 3-4: ArmIE instruction flow diagram

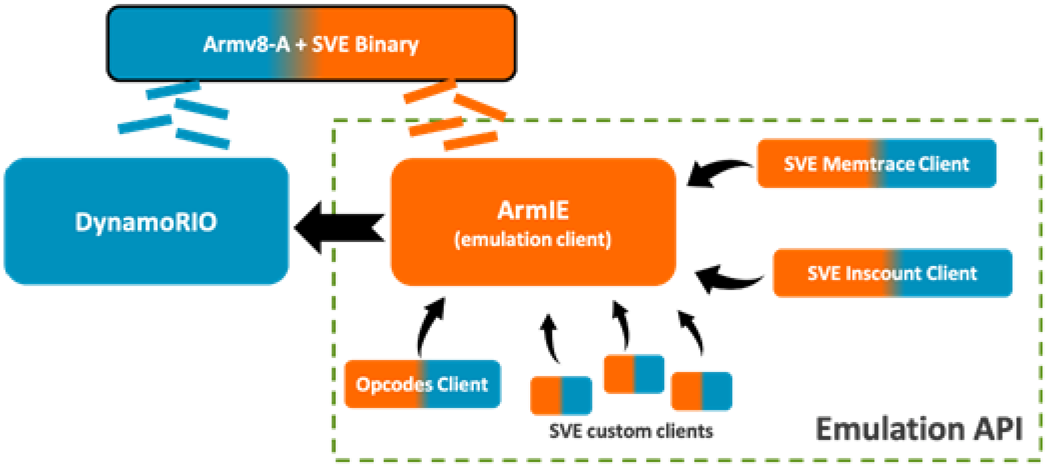

Armv8-A + SVE binary
Arm Instruction Emulator (ArmIE)
Converts unsupported instructions to native Armv8-A instructions
Linux
Armv8-A compatible Platform

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 3-5: 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.

Related information

Analyze Emulated SVE on Existing Armv8-A Hardware on page 69
Arm Instruction Emulator Developer and Reference Guide
Get started with Arm Instruction Emulator

3.5.2 Analyze Emulated SVE on Existing Armv8-A Hardware

This topic describes Arm® Instruction Emulator's structure, the clients Arm Instruction Emulator uses, and describes how to use Arm Instruction Emulator. The topic also describes the kind of analysis and studies that can be performed with the provided clients.

Note

This topic uses extracts of the Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE blog available on the Arm Community website.

The blog was published on November 5, 2018 and uses an older version of the Arm Instruction Emulator tool.

The case study is included to demonstrate the methodology used. The numbers presented might no longer be accurate.

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

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

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

Overview

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

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

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.

```
armie -msve-vector-bits=512 -i libinscount_emulated.so -- ./HACCKernels

Gravity Short-Range-Force Kernel (5th Order): 9178.27 -835.505 -167.99: 42.9214 s
205464290 instructions executed of which 167576110 were emulated instructions
```

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

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

```
$ARMIE_PATH/bin64/drrun -client $ARMIE_PATH/lib64/release/libsve_512.so 0 "" -client
 $ARMIE_PATH/samples/bin64/libinscount_emulated.so 1 "-only_from_app" -max_bb_instrs
 32 -max_trace_bbs 4 -- ./HACCKernels

Gravity Short-Range-Force Kernel (5th Order): 9178.27 -835.505 -167.99: 42.7263 s
201887951 instructions executed of which 167576110 were emulated instructions
```

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

Table 3-4: SVE Utilization (no shared libs) for different vector lengths

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

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
     5342564 : bcond
     5833081 : fmul
     8473704 : eor
77 unique emulated instructions written to undecoded.txt
```

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

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

 4150900 : 0xa5484c9b         ld1w   {z27.s}, p3/z, [x4, x8, lsl #2]
 4150900 : 0xa5484479         ld1w   {z25.s}, p1/z, [x3, x8, lsl #2]
 4150900 : 0xa5484458         ld1w   {z24.s}, p1/z, [x2, x8, lsl #2]
 4150900 : 0xa5484437         ld1w   {z23.s}, p1/z, [x1, x8, lsl #2]
 4150900 : 0x65b9033a         fmla   z26.s, p0/m, z25.s, z25.s
 4150900 : 0x65b8031b         fmla   z27.s, p0/m, z24.s, z24.s
 4150900 : 0x65b68359         fmad   z25.s, p0/m, z26.s, z22.s
 4150900 : 0x65b58358         fmad   z24.s, p0/m, z26.s, z21.s
 4150900 : 0x65b4837a         fmad   z26.s, p0/m, z27.s, z20.s
 4150900 : 0x65b3e35b         fnmsb  z27.s, p0/m, z26.s, z19.s
 4150900 : 0x65b2e35b         fnmsb  z27.s, p0/m, z26.s, z18.s
 4150900 : 0x65b1e35b         fnmsb  z27.s, p0/m, z26.s, z17.s
 4150900 : 0x65a6635b         fnmls  z27.s, p0/m, z26.s, z6.s
 4150900 : 0x65a58357         fmad   z23.s, p0/m, z26.s, z5.s
 4150900 : 0x659c0bbc         fmul   z28.s, z29.s, z28.s
 4150900 : 0x659c035a         fadd   z26.s, z26.s, z28.s
 4150900 : 0x659b0b5a         fmul   z26.s, z26.s, z27.s
 4150900 : 0x65970afa         fmul   z26.s, z23.s, z23.s
 4150900 : 0x65922343         fcmeq  p3.s, p0/z, z26.s, #0.0
 4150900 : 0x658da39d         fsqrt  z29.s, p0/m, z28.s
 4150900 : 0x658c80fc         fdivr  z28.s, p0/m, z28.s, z7.s
 4150900 : 0x6584035c         fadd   z28.s, z26.s, z4.s
 4150900 : 0x65834342         fcmge  p2.s, p0/z, z26.s, z3.s
 4150900 : 0x65820739         fsub   z25.s, z25.s, z2.s
 4150900 : 0x65810718         fsub   z24.s, z24.s, z1.s
 4150900 : 0x658006f7         fsub   z23.s, z23.s, z0.s
 4150900 : 0x25a91d01         whilelo        p1.s, x8, x9
 4150900 : 0x25834042         orr    p2.b, p0/z, p2.b, p3.b
```

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

Memory Tracing

The memory tracing client (memtrace) focuses on the dynamic memory accesses of the application, capturing information such as the accessed addresses and data sizes. Memtrace is based on the existing non-SVE DynamoRIO memtrace client, with added SVE emulation and tracing support. Running the emulated memtrace client results in two different memory trace files: an SVE-only trace and a non-SVE one. To keep the memory traces consistent, include an 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:

- Sequence Number
- Thread ID
- SVE Bundle
- isWrite (1 = write, 0 = read)

- Data Size (Bytes)
- Data Address
- PC

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

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

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

```
#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>
....
2990: 0, 0,  0,  4, 0x401a68, 0x401678
2991: 0, 0,  0,  4, 0x401a6c, 0x401680
2992: 0, 0,  0,  4, 0x401a70, 0x401688
2993: 0, 0,  0,  4, 0x401a74, 0x401690
2994: 0, 0,  0,  4, 0x401a78, 0x401698
2995: 0, 0,  0,  4, 0x401a7c, 0x4016a0
2996, 0, 0, 0, 64, 0x44c750, 0x4016f0
2997, 0, 0, 0, 64, 0x44d110, 0x4016f4
2998, 0, 0, 0, 64, 0x44dad0, 0x4016f8
2999, 0, 0, 0, 4, 0x44e490, 0x401750
3000, 0, 0, 0, 16, 0x44e49c, 0x401750
3001, 0, 0, 0, 4, 0x44e4b0, 0x401750
....
```

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

Memory traces are 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:

```
SVE vector size: 512 bits (64B)

Total Memory References = 20486551
   -> linear SVE: 16779970 (81.91%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 3706581 (18.09%)

Linear SVE accesses with at least 1 inactive lane:
   -> 1237914 (7.38% of linear SVE traces)
```

```
==============

Total Writes = 3121089  (34 unique writes - different PCs)
   -> linear SVE: 176536 (5.66%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 2944553 (94.34%)

Linear SVE Writes with at least 1 inactive lane:
   -> 3376 (1.91% of linear SVE write traces)

==============

Total Loads = 17365462 (51 unique loads - different PCs)
   -> linear SVE: 16603434 (95.61%)
   -> bundle SVE: 0 (0.00%)
   ->    non-SVE: 762028 (4.39%)

Linear SVE Loads with at least 1 inactive lane:
> 1234538 (7.44% of linear SVE loads)

==============

Distribution of memory operations:
   -> 15.23% Writes
   -> 84.77% Loads

SVE Bundles Stats:
   -> 0 SVE bundle accesses
```

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.

3.6 Tool quick references

This section collects some quick reference information for Arm® Compiler for Linux and Arm Forge.

3.6.1 Arm C/C++/Fortran Compiler Quick Reference

Quick reference for using Arm® C/C++/Fortran Compiler.

Common compiler options

For a full list of compiler options, see Arm C/C++ Compiler options or Arm Fortran Compiler options.

General

Table 3-5: Common compiler options for armclang, armclang++, and armflang

Option	Description
`-o <file>`	Write output to <file>.
`-c`	Only run pre-process, compile, and assemble steps.
`-g`	Generate source-level debug information.

Option	Description
`-Wall`	Enable all warnings.
`-w`	Suppress all warnings.
`-fopenmp`	Enable OpenMP.
`-On`	Level of optimization to use (`0`, `1`, `2`, or `3`).
`-Ofast`	Enables aggressive optimization of floating-point operations.
`-ffp-contract=<value>`	Form fused FP ops (e.g. FMAs): fast (fuses across statements disregarding pragmas) | on (only fuses in the same statement unless dictated by pragmas) | off (never fuses) | fast-honor-pragmas (fuses across statements unless dictated by pragmas). Default is 'fast' for CUDA, 'fast-honor-pragmas' for HIP, and 'on' otherwise.
`-fsave-optimization-record`	Enable the generation of a YAML optimization record file to use with Arm Optimization Report. Arm Optimization Report shows you the optimization decisions that the compiler is making, in-line with your source code, enabling you to better understand the unrolling, vectorization, and interleaving behavior.

Fortran

Table 3-6: armflang-specific Fortran options

Option	Description
`-cpp`	Preprocess Fortran files. Default for `.F`, `.F90`, and `.F95`.
`-module <path>`	Specifies a directory to place, and search for, module files.
`-Mallocatable=(95|03)`	`95`: Use Fortran 95 standard semantics for assignments to allocatables. `03`: Use Fortran 2003 standard semantics for assignments to allocatables.
`-fconvert=<setting>`	Set format for unformatted file access to numerical data to `big-endian`, `little-endian`, `swap`, or `native`.
`-r8`	Sets default `KIND` for real and complex declarations, constants, functions, and intrinsics to 64-bit (that is, real (`KIND=8`)). Unspecified real kinds are evaluated as `KIND=8`.
`-i8`	Set the default kind for `INTEGER` and `LOGICAL` to 64-bit (that is, `KIND=8`).

Pragmas

Arm C/C++ Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas use, and extend, the pragma `clang` loop directives.

```
#pragma clang loop vectorize(assume_safety)
```

Allows the compiler to assume that there are no aliasing issues in a loop.

```
#pragma clang loop unroll_count(_value_)
```

Forces a scalar loop to unroll by a given factor.

```
#pragma clang loop interleave_count(_value_)
```

Forces a vectorized loop to interleave by a given factor.

For more information about the pragma clang loop directives, see Auto-Vectorization in LLVM on the LLVM website, and Using pragmas to control auto-vectorization on the Arm Developer website.

Further resources

- Arm C/C++ Compiler Developer and Reference guide
- Get started with Arm C/C++ Compiler
- Arm Fortran Compiler Developer and Reference guide
- Get started with Arm Fortran Compiler

3.6.2 Arm Performance Libraries Quick Reference

Compiler command options that control the use of Arm® Performance Libraries (ArmPL).

Basic usage

To configure Arm Performance Libraries for your application, there are two decisions to make:

- **Decision 1: Do you want an OpenMP-enabled build?**

  To use serial Arm Performance Libraries:

  ```
  -armpl
  ```

  (defaults to no OpenMP)

  To use parallel Arm Performance Libraries:

  ```
  -armpl=parallel
  ```

  Note

  You can also enable the parallel Arm Performance Libraries using:

  ```
  -armpl -fopenmp
  ```

- **Decision 2: Do you need 32-bit or 64-bit integers?**

  To use 32-bit integers (the default):

  ```
  -armpl
  ```

To use 64-bit integers:

```
-armpl=ilp64
```

Note

- Options can be combined, for example:

  ```
  -armpl=ilp64, parallel
  ```

- If you are compiling Fortran code, you can also enable 64-bit integers using:

  ```
  -armpl -i8
  ```

Porting to Arm Performance Libraries

If your software uses standard BLAS, LAPACK, and FFTW interfaces, your code should port without problems.

If applicable, ensure that you include `armpl.h` rather than, for example, `mkl.h`.

Further resources

Arm Performance Libraries Reference Guide

3.6.3 Arm DDT Quick Reference

Quick reference for using Arm® DDT.

Workstation or remote interactive sessions

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Forge.

Note

The name of the environment variable is set by the system administrator. Check with your system administrator what the environment variable is called for your system.

2. Either, compile your application (including the `-g` flag), or locate an appropriately pre-compiled binary. To prepare the code and compile without optimizations, include the `-O0` optimization flag on the compile line:

```
mpicc -O0 -g myapp.c -o myapp.exe
```

Note

Turning off optimization flags is OPTIONAL. Optimization flags can reorder the code in unexpected ways and make application debugging less intuitive. If a bug only occurs within an optimized binary, keep the relevant optimizations.

3. Launch Arm DDT in interactive mode, use the Express Launch syntax:

```
ddt mpirun -n 8 ./myapp.exe arg1 arg2
```

4. Configure any advanced features for your job, such as memory debugging, in the **Run** dialog.
5. To start debugging, click **Run**.

Sessions on an HPC cluster with a job scheduler

To run Arm DDT in interactive mode, you can use the Arm Forge Remote Client or X-forwarding.

Note

For more information on using Arm Forge in non-interactive mode, see the Arm Forge user guide.

1. Start Arm Remote client, or an X-forwarding session. Either:
    - Arm Remote Client:
        a. Start Arm Remote Client.
        b. If it is your first time using the remote client, add the configuration details for your remote host.
        c. Select the connection from the **Remote Launch** drop-down menu.
    - X-forwarding session:
        a. Connect to the remote host system with X-forwarding enabled:

        ```
        ssh -X <remote-host>
        ```

        b. Prepare your environment. Load the environment module for Arm Forge.

        Note

        The name of the environment variable is set by the system administrator. Check with your system administator what the environment variable is called for your system.

2. On the login node, launch the Arm DDT debugger GUI:

```
ddt &
```

3. Either, compile your application (including the `-g` flag), or locate an appropriately pre-compiled binary. To prepare the code and compile without optimizations, include the `-O0` optimization flag on the compile line:

   ```
   mpicc -O0 -g myapp.c -o myapp.exe
   ```

   > **Note**
   >
   > Turning off optimization flags is OPTIONAL. Optimization flags can reorder the code in unexpected ways and make application debugging less intuitive. If a bug only occurs within an optimized binary, keep the relevant optimizations.

4. Edit your job script to run the *Reverse Connect* Arm DDT commands `ddt --connect`:

   ```
   ddt --connect mpirun -n 8 ./myapp.exe arg1 arg2
   ```

5. Submit your script.
6. When the GUI displays asking if you want to accept the incoming connection, click **Yes**.
7. Configure any advanced features for your job, such as memory debugging, in the **Run** dialog.
8. To start debugging, click **Run**.

Further resources

Arm Forge user guide

3.6.4 Arm MAP Quick Reference

Quick reference for using Arm® MAP.

Run Arm MAP on a workstation or remote interactive session

To run Arm MAP in non-interactive (*offline*) mode:

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Forge.

   > **Note**
   >
   > The name of the environment variable is set by the system administrator. Check with your system administator what the environment variable is called for your system.

2. Prepare the code and compile with optimizations:

   ```
   mpicc -O3 -g myapp.c -o myapp.exe
   ```

3. Generate a profile with the Express Launch syntax:

   ```
   map --profile mpirun -n 8 ./myapp.exe arg1 arg2
   ```

Note

In Arm MAP 19.x+ versions, you can profile python applications (sequential and parallel using mpi4py). To profile python applications, use:

```
map --profile python ./myapp.exe
```

4. Open the resulting `.map` file:

```
map ./myapp_8p_1n_YYYY-MM-DD_HH-MM.map
```

Note

The `.map` file can be opened anywhere, no compute node allocation is needed. However, you must tell the GUI where to look for the program source files.

Strategy

Serial comparison

Profile benchmark run, with codes compiled with both compilers, using map or perf.

Scale comparison

- Find a suitable benchmark to do scaling runs across various numbers of cores (weak and strong scaling).
- Consider placement of tasks to minimize interference between tasks.

Look at whether hot sections have similar work between compilers

- To locate sections of code that consume most of the run-time, use Arm MAP profile runs.
- Are the locations the same for both compilers? Consider if these sections could be improved.
- Report major run-time differences across compilers to the Arm C/C++/Fortran Compiler team.

Search for compiler specific sections/intrinsics

- Search the source for compiler-specific intrinsics, or pragmas, and consider if these can be ported to Arm C/C++/Fortran Compiler or platform-compatible versions.
- If AVX512 vector instructions are present, consider converting them.

Further resources

Arm Forge user guide

3.6.5 Arm Performance Reports Quick Reference

To start Arm® Performance Reports:

1. Log in to a terminal session and prepare your environment. Load the environment module for Arm Performance Reports.

   Note

   The name of the environment variable is set by the system administrator. Check with your system administrator what the environment variable is called for your system.

2. Prepare the code and compile with optimizations:

```
mpicc -O3 -g myapp.c -o myapp.exe
```

3. Generate your performance report with the Express Launch syntax:

```
perf-report mpirun -n 8 ./myapp.exe arg1 arg2
```

4. Open the resulting `.html` or `.txt` file.

Further resources

Arm Performance Reports user guide

3.7 Fortran compiler migration guides

To assist Fortran developers who are familiar with using the `gfortran`, `ifort`, and `pgfortran` compilers, this chapter provides an overview of `armflang`, and discusses the differences between each compiler and `armflang`.

3.7.1 Overview of Arm Fortran Compiler (armflang)

This topic introduces Arm® Fortran Compiler.

For more information on Arm Fortran Compiler, see the Arm Fortran Compiler Reference Guide or Arm Fortran Compiler product web page.

Invoking Arm Fortran Compiler

To invoke Arm Fortran Compiler for preprocessing, compilation, assembly, and linking, use `armflang`.

To access compiler details and documentation, use:

Table 3-7: GNU and Arm Compiler commands

	Arm
Version details	`armflang --version`

Help and documentation	`armflang --help`
	`man armflang`

Supported file types

The extensions `.f90`, `.f95`, `.f03`, and `.f08` are used for modern, free-form source code that conforms to the Fortran 90, Fortran 95, Fortran 2003, or Fortran 2008 standards.

The extensions `.F90`, `.F95`, `.F03`, and `.F08` are used for source code that requires preprocessing, and which is preprocessed automatically.

It is possible to instruct `armflang` to preprocess source irrespective of file extension by using the `-cpp` option, as detailed in the next section.

Typically, `.f` and `.for` extensions are used for older, fixed-form code, such as FORTRAN77.

Arm hardware options

GCC and Arm C/C++/Fortran Compiler, have three hardware compiler options in common: `-march`, `-mtune`, and `-mcpu`:

- `-march=X`: Tells the compiler that X is the minimal architecture the binary must run on. The compiler is free to use architecture-specific instructions. This option behaves differently on Arm and x86. On Arm, `-march` does not override `-mtune`, but on x86 `-march` does override both `-mtune` and `-mcpu`.
- `-mtune=X`: Tells the compiler to optimize for microarchitecture X, but does not allow the compiler to change the ABI or make assumptions about available instructions. This option has the more-or-less the same meaning on Arm and x86.
- `-mcpu=X`: On Arm, this option is a combination of `-march` and `-mtune`. It simultaneously specifies the target architecture and optimizes for a given microarchitecture. On x86, this option is a deprecated synonym for `-mtune`.

GCC and Arm C/C++/Fortran Compiler support passing the special parameter value `native` to these options. The `native` value tells the compiler to automatically detect the architecture or microarchitecture of the machine on which the compiler is executing.

Note

Arm C/C++/Fortran Compiler does not support the use of `-march=native`. To aid portability, GCC on AArch64 does support the use of `-march=native`.

These compiler options control binary code generation. Correctly using these options can greatly improve run-time performance. If you are not cross compiling, the simplest and easiest method to get the best performance on Arm, with both GCC and LLVM-based compilers, is to only use `-mcpu=native`, and actively avoid using `-mtune` or `-march`.

Note

Automatic detection of the architecture and processor is independent of the optimization level that is denoted by the `-On` option and similar options, as detailed in the **Commonly used options** and **Optimization compiler options** sections in each compiler guide.

Optimized math functions with Arm Performance Libraries

Arm Performance Libraries (ArmPL) provide the following optimized standard core math libraries for high-performance computing applications on Arm processors:

- BLAS: Basic Linear Algebra Subprograms (including XBLAS, the extended precision BLAS).
- LAPACK: A comprehensive package of higher level linear algebra routines. To find out what the latest version of LAPACK that is supported in Arm Performance Libraries is, see Arm Performance Libraries.
- FFT functions: a set of Fast Fourier Transform routines for real and complex data using the FFTW interface.
- Sparse linear algebra.
- libamath: a subset of libm, which is a set of optimized mathematical functions.

`armpl` provides a simple interface for selecting thread-parallelism and architectural tuning. Arm Performance Libraries also provides optimized versions of the C mathematical functions, tuned scalar and vector implementations of Fortran math intrinsics, and auto-vectorization of mathematical functions.

Combining `-armpl` with the `-mcpu=native|<target>` or `-march=<architecture>` `armflang` options enables the compiler to find the appropriate Arm Performance Libraries header files during compilation, and the compatible libraries during linking. Both `-armpl` and either the `-mcpu` or `-march` options are required to link to the correct version of Arm Performance Libraries.

Note

- If your build process compiles and links as two separate steps, ensure that you add the same `-armpl` and `-m{cpu|arch}` options to both the compile and link commands.
- If you compile (and link) on a target that is different to the target where you will run the application, compile the program so that it is suitable to run on any Armv8-A-based or Armv8-A-based SVE system, using `-march=armv8a` or `-march=armv8-a+sve`, respectively. You can also use `-march=armv8-a+sve` if you will use SVE emulation software, for example Arm Instruction Emulator, to run the program on an Armv8-A-based system.

For more information on Arm Performance Libraries, see Get started with Arm Performance Libraries.

Compiler directives

Directives are used to provide additional information to the compiler, and to control the compilation of specific code blocks, for example, loops. The Arm Fortran Compiler supports the following common directives:

Table 3-8: Arm Fortran Compiler directives

Directive	Usage	Description

`IVDEP`	`!DIR$ IVDEP`
	`<do loop>`
		A generic directive which forces the compiler to ignore any potential memory dependencies of iterative loops, and to vectorize the loop.

`OMP SIMD`	`!$OMP SIMD`
	`<do loop>`
		An OpenMP directive to indicate that a loop can be transformed into a Single Instruction Multiple Data (SIMD) loop.
		Note:
		• -fopenmp must be set.
		• There is no support for `OMP SIMD` clauses.

`PREFETCH`	`!$MEM PREFETCH <var_1>[,<var_2>[,...]]`	Tells the compiler to generate prefetch instructions to fetch elements and load them in the data cache, ahead of their first use. Users can provide a prefetch distance. Prefetching elements can improve performance by reducing main memory latency.

`VECTOR ALWAYS`	`!DIR$ VECTOR ALWAYS`
	`<do loop>`
		Forces the compiler to vectorize a loop, and ignores any potential performance implications.
		Note:
		The loop must be vectorizable.

`NOVECTOR`	`!DIR$ NOVECTOR`
	`<do loop>`
		Disables the vectorization of a loop.

`UNROLL`	`!DIR$ UNROLL`
	`<do loop>`
		Instructs the compiler optimizer to unroll a DO loop when optimization is enabled with the compiler optimization options `-O2` or higher.

Generating Position Independent Code (PIC) with fPIC on AArch64

The generation of Position Independent Code (PIC) is typically required for building shared libraries. Supplying the command line option `-fPIC` at compile time instructs `armflang` to generate Position Independent Code (PIC), and is generally consistent with the behavior of other compilers.

Note

PGI compilers do not differentiate between `-fPIC` and `-fpic` which are documented as interchangeable on x86 architectures. For more information on migrating from the PGI `pgfortran` compiler to Arm Compiler, see armflang for pgfortran users.

However, while the use of `-fpic` is often interchangeable with `-fPIC` on x86, it is not the case with GCC on AArch64. `-fpic` uses an address mode with a smaller number of entries in the Global Offset Table. As a result, `-fpic` is not considered to be portable between x86_64 and AArch64 architectures.

Allocating stack variables

- **Thread-safe recursion**

  The `-frecursive` option allocates all local variables on the stack. This allows thread-safe recursion and is applied implicitly for source compiled with the `-fopenmp` option.

  Use the `-frecursive` options when compiling a procedure that:

  - Has no OpenMP elements and is not compiled using the `-fopenmp` option.
  - Is called from within an OpenMP parallel region in source and is compiled with the `-fopenmp` option.

- **Automatic arrays**

  This feature of Fortran 2003 allows allocatable arrays to be allocated, and dynamically resized without the need for calls to `ALLOCATE` and `DEALLOCATE`. Automatic arrays are stored on the heap, regardless of the `-frecursive` option, unless `-fstack-arrays` is specified.

Note

Use of the stack for local variables and automatic arrays can have implications for the stack size. To avoid running out of stack, it might be necessary to increase the stack size. For example, to remove the stack-size limit, enter `ulimit -s unlimited` at the command line.

Line lengths

The Fortran standard for free-form source (from Fortran90 onwards) sets a maximum line length of 132 characters. Statements can be broken over a maximum of 255 lines using the ampersand, &, continuation mark. Many compilers permit the use of lines longer than 132 characters.

`armflang` limits line lengths to 2100 characters and generates a compile time error if there are source lines, including comments, longer than 2100 characters. To compile with Arm Fortran Compiler, you must ensure that all source lines are within this limit.

Note

Arm C/C++/Fortran Compiler versions that are earlier than 23.04 limited line lengths to 264 characters. Using compiler macros in versions that are earlier than 23.04 can lead to the generation of source lines longer than 264 characters at compile time.

Language extensions

There are several common extensions to the Fortran language which are typically supported by many existing compilers, generally for legacy reasons, including armflang. Often, the required functionality is now part of the language standard, even though it uses a different syntax. The following table shows common language extensions and their standards-compliant alternatives, where available.

Table 3-9: Fortran language extensions and their standard-compliant alternatives

Extension	Purpose	Standard-compliant alternative	Notes
`IARGC()`	Function call which returns the number of command line arguments supplied	`COMMAND_ARGUMENT_COUNT()`	Introduced with 2003 standard.
`ISNAN(x)`	Logical function returns `.TRUE.` if the REAL argument x is Not-a-Number (NaN).	`IEEE_IS_NAN(x)`	Introduced with 2003 standard. Requires IEEE_ARITHMETIC module.
`GETARG(pos, arg)`	Subroutine call which returns the `pos`-th argument that is passed on the command line when the program was invoked, and returns it as arg.	`GET_COMMAND_ARGUMENT(pos,arg, len, status)`	Introduced with 2003 standard. `arg`, `len`, and `status` are OPTIONAL arguments.
`GETENV(name, arg)`	Subroutine call which returns the environment variable name as `arg`.	`GET_ENVIRONMENT_VARIABLE(name, arg, len, status, trim_name)`	Introduced with 2003 standard. `arg`, `len`, and `status` are OPTIONAL arguments.
`GETCWD(dir, status)`	Subroutine call which returns the current working directory as dir. `status` is an OPTIONAL argument which returns 0 on success, and a nonzero error code when not successful.	No equivalent functionality at present.	No equivalent functionality in the 2003 standard.

For more information on supported language extensions, see the Fortran intrinsics chapter in the Arm Fortran Compiler Reference Guide.

Pre-defined macros

`armflang` has the following compiler and machine-specific predefined macros:

Table 3-10: Pre-defined macros

Macro	Value	Purpose
`__ARM_ARCH`	INTEGER	Defined as an integer value and expands to 8 to indicate that the system implements v8 of the Armv8-A architecture. Selection of architecture-dependent source at compile time.
`__ARM_LINUX_COMPILER__`	1	Defined as an integer value and expands to 1 to indicate Arm Compiler for Linux.
`__ARM_LINUX_COMPILER_BUILD__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux build number.

Macro	Value	Purpose
`__armclang_major__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux major version number.
`__armclang_minor__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux minor version number.
`__armclang_version__`	STRING	Defined as a string value and expands to the full Arm Compiler for Linux version number.
`__FLANG`	1	Defined as an integer value and expands to 1 to indicate a FLANG-derived compiler. `__FLANG` is often included in Makefiles that support flang compilers.

Note

The preceding list is not exhaustive, to read about more predefined macros that are available to `armflang`, see: https://developer.arm.com/documentation/101380/latest/Fortran-language-reference/Predefined-macro-support.

Detailed compiler options

Passing the option `-###` to `armflang` causes it to print the complete options used at each stage of the compilation, without executing them.

Understand the optimization choices the compiler makes

Arm C/C++/Fortran Compiler incorporates two tools to help you better understand the optimization decisions that it makes:

Arm Optimization Report

Arm Optimization Report is a new feature of Arm Compiler for Linux version 20.0 that builds upon the llvm-opt-report tool available in open-source LLVM. The new Arm Optimization Report feature makes it easier to see what optimization decisions the compiler is making about unrolling, vectorizing, and interleaving, in-line with your source code.

To enable Arm Optimization Report:

1. At compile time, add the `-fsave-optimization-record` to the command line.

   A `<filename>.opt.yaml` report is generated by the compiler, where `<filename>` is the name of the binary.

2. Use Arm Optimization Report (`arm-opt-report`) to inspect the `<filename>.opt.yaml` report as augmented source code:

   ```
   arm-opt-report <filename>.opt.yaml
   ```

   The annotated source code appears in the terminal.

For more information, refer to the Arm Optimization Reports documentation in the Arm Fortran Compiler reference guide.

Optimization Remarks

Optimization Remarks can be used to see which code has been inlined or to understand why a loop has not been vectorized.

To enable Optimization Remarks, pass one or more of the following `-Rpass` options at compile time:

Table 3-11: Options to enable Optimization Remarks

`-Rpass` options	Description
`-Rpass=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has optimized.
`-Rpass-analysis=<regexp>`	To request information about what Arm C/C++/Fortran Compiler has analyzed.
`-Rpass-missed=<regexp>`	To request information about what Arm C/C++/Fortran Compiler failed to optimize.

In each case, `<regexp>` is used to select the type of remarks to provide. For example, loop-vectorize for information on vectorization, and inline for information on in-lining, or `.*` to report all optimization remarks. `Rpass` accepts regular expressions, so (loop-vectorize|inline) can be used to capture any remark on vectorization or inlining.

For example, to get actionable information on which loops can and cannot be vectorized at compile time, pass:

```
-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize
```

Note

- Optimization Remarks are piped to `stdout` at compile time.

For more information on optimization remarks, see the Fortran compiler reference guides.

Further resources

Standards compliance

Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler

- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

3.7.2 armflang for gfortran users

The reference versions used in this guide are:

- GCC (`gfortran` 8.2.0)
- Arm® Fortran Compiler

Invoking the compiler

The following table gives the equivalent GCC and Arm C/C++/Fortran Compiler commands to invoke Arm Fortran Compiler for preprocessing, compilation, assembly, and linking.

Table 3-12: Invoking the compiler

GCC	Arm
`gfortran <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent GCC and Arm C/C++/Fortran Compiler commands to access compiler details and documentation.

Table 3-13: Accessing version information and documentation

	GCC	Arm
Version details	`gfortran --version`	`armflang --version`

Help and documentation	`gfortran --help`
	`man gfortran`
		`armflang --help`
		`man armflang`

Commonly used flags

The following table summarizes some of the compiler options most commonly used with GCC and gives the equivalent options to use with the Arm Fortran Compiler:

Table 3-14: GCC and Arm Compiler equivalent options

GCC	Arm	Description
`-c`	`-c`	Run only preprocess, compile, and assemble steps.
`-o` *filename*	`-o` *filename*	Write to output filename.
`-g`	`-g`	Generate source level debug information.

`-Wall`
`-warn none`
	`-Wall`
	`-w`
		Enable all warnings.
		Suppress all warnings.

GCC	Arm	Description
`-cpp` `-nocpp`	`-cpp` `-nocpp`	Preprocess Fortran source files. Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, `.F`, `.F90`, `.F95`, `.F03` and `.F08` are preprocessed. `-cpp` forces the compiler to use the processor for all source files.
`-fopenmp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-J` *path* `-I` *path*	`-module` *path*	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See the Compiler options for the Arm Fortran Compiler.
`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is allocated (or reallocated) to match the dimensions of the right-hand side. `-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions. **Note:** Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: allocation (or reallocation) on assignment. By default, earlier versions of `armflang` do not support this feature.

GCC	Arm	Description

`-byteswapio`	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big-endian from little-endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big-endian, little-endian, or native.
		Note:
		Default behavior is native.

`-D`*macro=value*	`-D`*macro=value*	Set macro to *value*.
`-L`*directory*	`-L`*directory*	Add *directory* to the include search path.
`-l`*lib*	`-l`*lib*	Search for the library *lib* when linking.
`-fdefault-real-8`	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64-bit (such as real (`KIND=8`)). Unspecified real kinds are evaluated as KIND=8.
`-fdefault-integer-8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64-bit (KIND=8).
`-frecord-marker=n`	N/A	Length of record markers for unformatted files. `n` can be 4 or 8. Default is 4. However, older versions of gfortran default to 8. `armflang` uses a record marker of length 4 bytes.
`-fpic` `-fPIC`	`-fpic` `-fPIC`	Generate Position Independent Code (PIC). Using the `-fpic` compiler flag with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and use a Global Offset Table (GOT) limited to 32kiB. Arm C/C++/Fortran Compiler treats `-fpic` as equivalent to `-fPIC`. To increase code portability, Arm recommends using `-fPIC` when compiling with Arm C/C++/Fortran Compiler. For more information on the use of `-fpic` and `-fPIC` on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by GCC and Arm Fortran Compiler:

Table 3-15: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	`-On`	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.

Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance.
		With `armflang`, this is equivalent to:
		• Setting: `-O3 -Menable-no-infs`
		`-Menable-no-nans`
		`-Menable-unsafe-fp-math`
		`-fno-signed-zeros -freciprocal-math`
		`-fno-trapping-math -ffp-contract=fast`
		`-ffast-math -ffinite-math-only`
		`-fstack-arrays`
		• Unsetting: `-fmath-errno`

Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • `fast` = always on (default for `-O1` and above) • `off` = never
Reduced floating-point precision	`-ffast-math` `-funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.

Description	Syntax	Notes
Finite maths	`-ffinite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

Language extensions

For information on supported language extensions, see the Arm Fortran Compiler Reference Guide.

Fortran formatted I/O

`armflang` adopts the Linux/UNIX convention of using the line-feed character (`'LF'`, `'0x0A'`, `'\n'`) as the record terminator in formatted I/O, for both read and write operations. However, gfortran also accepts the carriage return character (`'CR'`, `'0x0D'`, `'\r'`) to denote the end of records on read operations. This can lead to differing behavior between `armflang` and gfortran builds when accessing files containing 'CR' characters, such as text files generated on Windows platforms, which use 'CR-LF' to denote the end of lines.

Further resources

Standards compliance

Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

3.7.3 armflang for ifort users

The reference versions used in this guide are:

- Intel Fortran Compiler 17.0.1
- Arm® Fortran Compiler

Invoking the compiler

The following table gives the equivalent Intel and Arm C/C++/Fortran Compiler commands to invoke the Fortran compiler for preprocessing, compilation, assembly, and linking.

Table 3-16: Invoking the compiler

Intel	Arm
`ifort <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent Intel and Arm C/C++/Fortran Compiler commands to access compiler details and documentation.

Table 3-17: Accessing version information and documentation

	Intel	Arm
Version details	`ifort --version`	`armflang --version`

Help and documentation	`ifort --help`
	`man ifort`
		`armflang --help`
		`man armflang`

Commonly used flags

The following table summarizes some of the compiler options most commonly used with the Intel Fortran compiler and gives the equivalent options to use with the Arm Fortran Compiler:

Table 3-18: Intel and Arm Compiler equivalent options

Intel	Arm	Description
`-c`	`-c`	Run only preprocess, compile, and assemble steps.
`-o` *filename*	`-o` *filename*	Write to output filename.
`-g`	`-g`	Generate source level debug information.

`-warn all`
`-warn none`
	`-Wall`
	`-w`
		Enable all warnings.
		Suppress all warnings.

`-fpp` `-nofpp`	`-cpp` `-nocpp`	Preprocess Fortran source files. Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, `.F`, `.F90`, `.F95`, `.F03` and `.F08` are preprocessed. `-cpp` forces the compiler to use the processor for all source files.
`-qopenmp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-module` *path*	`-module` *path*	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See the Compiler options for the Arm Fortran Compiler.

Intel	Arm	Description
`-standard-realloc-lhs` `-nostandard-realloc-lhs`	`-frealloc-lhs` `-fno-realloc-lhs`	`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is (re)allocated to match the dimensions of the right-hand side. `-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions. **Note:** Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: (re)allocation on assignment. By default, earlier versions of `armflang` do not support this feature.

`-convert big-endian-convert little-endian-convert native`	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big-endian from little-endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big-endian, little-endian, or native.
		Note:
		Default behavior is native.

`-D`*macro*`=`*value*	`-D`*macro*`=`*value*	Set *macro* to value.
`-L`*directory*	`-L`*directory*	Add *directory* to the include search path.
`-l`*lib*	`-l`*lib*	Search for the library *lib* when linking.

`-real-size 64`
`-r8`
	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64-bit (such as real (KIND=8)).
		Unspecified real kinds are evaluated as KIND=8.

`-integer-size 64`
`-i8`
	`-i8`	Set the default kind for INTEGER and LOGICAL to 64-bit (KIND=8).

Intel	Arm	Description
-fpic -fPIC	-fpic -fPIC	Generate Position Independent Code (PIC). Arm C/C++/Fortran Compiler and ICC both treat -fpic as equivalent to -fPIC. To increase code portability, Arm recommends using -fPIC when compiling with Arm C/C++/Fortran Compiler. For more information on the use of -fpic and -fPIC on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by the Intel and Arm Fortran Compiler:

Table 3-19: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	-On	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.

Description	Syntax	Notes

Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance.
		With `armflang`, this is equivalent to:
		• Setting: `-O3 -Menable-no-infs`
		`-Menable-no-nans`
		`-Menable-unsafe-fp-math`
		`-fno-signed-zeros -freciprocal-math`
		`-fno-trapping-math -ffp-contract=fast`
		`-ffast-math -ffinite-math-only`
		`-fstack-arrays`
		• Unsetting: `-fmath-errno`

Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • `fast` = always on (default for `-O1` and above) • `off` = never
Reduced floating-point precision	`-ffast-math` `-funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.
Finite maths	`-ffinite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

Handling backslash characters

The default behavior in `armflang` is for backlash () to be treated as a special character; this is not the case for ifort.

To make `armflang` match ifort's behavior, use `-fno-backslash`.

To make ifort match armflang's behavior use `-assume bscc`.

Further resources

Standards compliance

Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

3.7.4 armflang for pgfortran users

The reference versions used in this guide are:

- PGI Fortran Compiler 18.5
- Arm® Fortran Compiler

Invoking the compiler

The following table gives the equivalent PGI and Arm C/C++/Fortran Compiler commands to invoke the Fortran compiler for preprocessing, compilation, assembly, and linking.

Table 3-20: Invoking the compiler

PGI	Arm
`pgfortran <options> <filename>`	`armflang <options> <filename>`

The following table gives the equivalent PGI and Arm C/C++/Fortran Compiler commands to access compiler details and documentation.

Table 3-21: Accessing version information and documentation

	PGI	Arm
Version details	`pgfortran --version`	`armflang --version`

Help and documentation	`pgfortran --help`
	`man pgFortran`
		`armflang --help`
		`man armflang`

Commonly used flags

The following table summarizes some of the compiler options most commonly used with the PGI Fortran compiler and gives the equivalent options to use with the Arm Fortran Compiler:

Table 3-22: PGI and Arm Compiler equivalent options

PGI	Arm	Description
`-c`	`-c`	Run only preprocess, compile, and assemble steps.
`-o` *filename*	`-o` *filename*	Write to output filename.
`-g`	`-g`	Generate source level debug information.

PGI	Arm	Description
`-Minform=inform`	`-Wall`	Enable all warnings.

`-w`
`-silent`
`-Minform=severe`
	`-w`	Suppress all warnings.

`-Mpreprocess`	`-cpp`	Preprocess Fortran source files.
	`-nocpp`	Do not preprocess Fortran source files. **Note:** By default, source files with the extensions, `.F`, `.F90`, `.F95`, `.F03` and `.F08` are preprocessed. `-cpp` forces the compiler to use the processor for all source files.
`-mp`	`-fopenmp`	Enable OpenMP. See OpenMP support for Arm Fortran Compiler.
`-module` *path*	`-module` *path*	Specifies a directory to place and search for module files.
`-On`	`-On`	Level of optimization to use, where n=0,1,2,3. See the Compiler options for the Arm Fortran Compiler.

PGI	Arm	Description

`-Mallocatable=03`
`-Mallocatable=95`
	`-frealloc-lhs`
	`-fno-realloc-lhs`
		`-frealloc-lhs` uses Fortran 2003 standard semantics for assignments to allocatables. An allocatable object on the left-hand side of an assignment is (re)allocated to match the dimensions of the right-hand side.
		`-fno-realloc-lhs` uses pre-Fortran 2003 standard semantics for assignments to allocatables. The left-hand side of an allocatable assignment is assumed to be allocated with the correct dimensions. Incorrect behavior can occur if the left-hand side is not allocated with the correct dimensions.
		Note:
		• Default behavior in `armflang` versions 19.0+ supports the Fortran 2003 standard feature: (re)allocation on assignment. By default, earlier versions of `armflang` do not support this feature.
		• In version 19.0 of armflang, `-Mallocatable=03` and `-Mallocatable=95` are supported instead of `-frealloc-lhs` and `-fno-realloc-lhs`, respectively. The `-Mallocatable` option remains supported in the armflang, but from versions 23.04+ the documentation refers to the `-frealloc-lhs` and `-fno-realloc-lhs` nomenclature.

`-byteswapio`	`-fconvert=big-endian`
	`-fconvert=little-endian`
	`-fconvert=native`
	`-fconvert=swap`
		Swap the byte ordering for unformatted file access of numeric data to big-endian from little-endian, or the other way round.
		`armflang` also provides options to set the byte order explicitly to big-endian, little-endian, or native.
		Note:
		Default behavior is native.

`-D`*macro*=*value*	`-D`*macro*=*value*	Set *macro* to value.
`-L`*directory*	`-L`*directory*	Add *directory* to the include search path.
`-l`*lib*	`-l`*lib*	Search for the library *lib* when linking.

PGI	Arm	Description
`-r8`	`-r8`	Set the default KIND for real and complex declarations, constants, functions, and intrinsics to 64-bit (such as real (KIND=8)). Unspecified real kinds are evaluated as KIND=8.
`-i8`	`-i8`	Set the default kind for INTEGER and LOGICAL to 64-bit (KIND=8).
`-fpic` `-fPIC`	`-fpic` `-fPIC`	Generate Position Independent Code (PIC). Arm C/C++/Fortran Compiler and PGI both treat `-fpic` as equivalent to `-fPIC`. To increase code portability, Arm recommends using `-fPIC` when compiling with Arm C/C++/Fortran Compiler. For more information on the use of `-fpic` and `-fPIC` on AArch64, see the Note about building Position Independent Code PIC on AArch64.

Optimization compiler options

The following table summarizes some of the most commonly used compiler options provided by the PGI and Arm Fortran Compiler:

Table 3-23: Commonly used optimization options

Description	Syntax	Notes
Basic optimization switches	`-On`	Optimization level where n=0,1,2,3. There is no direct correlation between the optimizations employed at each level between the two compilers. At n=0, the compiler performs little or no optimization. At n=3, the compiler performs aggressive optimization. At n=2 and n=3, debug information might not be satisfactory because the mapping of object code to source code is not always clear and the compiler can perform optimizations that cannot be described in the debug information.

Description	Syntax	Notes

Aggressive optimization	`-Ofast`	Enables all `-O3` optimizations from level 3 and performs aggressive optimization, which can violate strict language compliance.
		With `armflang`, this is equivalent to:
		• Setting: `-O3 -Menable-no-infs`
		`-Menable-no-nans`
		`-Menable-unsafe-fp-math`
		`-fno-signed-zeros -freciprocal-math`
		`-fno-trapping-math -ffp-contract=fast`
		`-ffast-math -ffinite-math-only`
		`-fstack-arrays`
		• Unsetting: `-fmath-errno`

Fused floating-point operations	`-ffp-contract=fast/off`	Instructs `armflang` to perform fused floating-point operations, such as fused multiply adds. • `fast` = always on (default for `-O1` and above) • `off` = never
Reduced floating-point precision	`-ffast-math` `-funsafe-math-optimizations`	Allows aggressive, lossy, floating-point optimizations. Allows reciprocal optimizations and does not honor trapping or signed zero.
Finite maths	`-ffinite-maths-only`	Enable optimizations that ignore the possibility of NaNs and Infs.

Further resources

Standards compliance

Arm provides full support for Fortran 2003 and prior standards, and partial support for Fortran 2008.

Additional information

- Arm Fortran Compiler Developer and Reference Guide
- Statement support in Arm Fortran Compiler
- Intrinsic support in Arm Fortran Compiler
- OpenMP support in Arm Fortran Compiler
- Arm Performance Libraries

3.8 Porting and Tuning Recipes

Describes where to find recipes for building many common HPC applications.

For detailed, community-driven instructions on how to build many common scientific applications, benchmarks, and libraries using both open-source tools and the Arm HPC tools, see the Packages wiki.

3.9 Porting 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).

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

- Porting and Optimizing HPC Applications for Arm guide. This guide discusses the tools Arm offers to help you port your codes to Arm (Neon®). It includes some introductory information about Neon, illustrates some example code, and discusses the tooling Arm provides to prepare your code for Arm Neon-based hardware.
- Community-driven porting recipes on the Arm HPC Packages Wiki. The GitLab repository contains a list of ported application recipes which the Arm HPC ecosystem 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 latest presentations delivered at these events.
- Arm Compiler for Linux documentation.
- HPC blogs and HPC Forum. 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.

4. Coding for Scalable Vector Extension (SVE)

This topic describes how to code for Scalable Vector Extension (SVE)-enabled Arm-based processors.

Many of the examples compare Arm® Neon® code to Arm SVE code to illustrate the difference between writing static and scalable vector code. Both sets of examples are written for an Arm-based target, however if you are migrating your code from another architecture, the concepts behind how the code changes can still be used as a guide of how to change your code to prepare it for SVE-enabled Arm-based processors.

The chapter also introduces the Vector Length Agnostic (VLA) programming model that allows your code to adapt to the available vectors on the machine you run your code on, and provides a number of generic vector and matrix multiplication examples.

4.1 SVE Vector Length Agnostic (VLA) 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.

4.1.1 Loop vectorization and predication

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

Example 1: Loop vectorization

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:

```
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**):

```
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 (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**):

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

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

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

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

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:

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

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

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

```
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**):

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

Example 2: 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):

```
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**):

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

Note

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

Example 3: 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.

example03_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;
}
```

example03_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
```

Example 4: 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):

```
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**):

```
     mov     x5, 0
     b       cond
loop:
     ld1w    z1.s, p0/z, [x4, x5, lsl 2]
     ld1w    z0.s, p0/z, [x1, z1.s, sxtw 2] # load a vector
                                           # from  'x1 + sxtw(z1.s) << 2'
     ld1w    z1.s, p0/z, [x2, x5, lsl 2]
```

```
      add     z0.s, p0/m, z0.s, z1.s
      st1w    z0.s, p0, [x0, x5, lsl 2]
      incw    x5
cond:
      whilelt p0.s, x5, x3
      b.first    loop
      ret
```

Related information

Arm C Language Extensions for SVE

4.1.2 For and While loop vectorization

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

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

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

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

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

Note

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

For and 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
```

```

.L_vector_loop:
    LD1 {v0.4s}, [x1], #16
    LD1 {v1.4s}, [x2], #16
    ADD v5.4s, v0.4s, v1.4s
    ST1 {v5.4s}, [x0], #16
    SUB w6, w6, #4
    CBNZ w6, .L_vector_loop

.L_tail:
    AND w3, w3, #3
    CBZ w3, .L_end

.L_scalar_loop:
    LDRSW w9, [x1], #4
    LDRSW w10, [x2], #4
    ADD w11, w9, w10
    STR w11, [x0], #4
    SUB w3, w3, #1
    CBNZ w3, .L_scalar_loop

.L_end:
    RET
```

For and While loop SVE vectorization

The SVE Vector Length Agnostic (VLA) vectorization approach involves carefully setting the predicates to manage register partitioning, predicate handling, loop counter, and pointer offset updates over loop iterations, with the help of specific loop control instructions. The SVE vectorized code is shorter and more compact than the Advanced SIMD vectorized code. The vectorized loop is seamlessly terminated, and there is no need for the scalar loop.

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

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

In the vectorized loop at lines 7-14, the governing predicate controls the active lanes of the load, addition, and store instructions. Only valid data are:

1. Accessed from the memory and loaded into the vector registers (lines 8-9).
2. Processed (line 10).

3. Stored to the memory (line 11).

The inactive lanes of vector registers are zeroed by load instructions (lines 8-9).

The value in register `x9` is used both as the loop counter and as the load and store pointer offset. Each vectorized loop iteration is incremented in the vector length agnostic approach. With the instruction `INCW`, this value is incremented by the number of 32-bit elements in the implemented vector length (line 12).

Note

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

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

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

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

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

For and 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.

```
    PTRUE p0.s
    PRFW PLDL1STRM, p0,[x1]
    PRFW PLDL1STRM, p0,[x1, #1, MUL VL]
    PRFW PLDL1STRM, p0,[x1, #2, MUL VL]
    PRFW PLDL1STRM, p0,[x2]
    PRFW PLDL1STRM, p0,[x2, #1, MUL VL]
    PRFW PLDL1STRM, p0,[x2, #2, MUL VL]

    MOV w3, w3
    MOV x9, #0
```

```
11       ADDVL x10, x1, #3
12       ADDVL x12, x2, #3
13
14       WHILELT p1.s, x9, x3
15       B.NONE .L_return
16
17   .L_loopStart:
18       LD1W z1.s, p1/Z, [x1, x9, LSL #2]
19       LD1W z2.s, p1/Z, [x2, x9, LSL #2]
20       PRFW PLDL1STRM, p0,[x10, x9, LSL #2]
21       PRFW PLDL1STRM, p0,[x12, x9, LSL #2]
22       ADD z1.s, p1/M, z1.s, z2.s
23       ST1W z1.s, p1, [x0, x9, LSL #2]
24       INCW x9
25       WHILELT p1.s, x9, x3
26       B.FIRST .L_loopStart
27
28   .L_return:
29       RET
```

In the 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.

4.1.3 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:

```
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.

```
        PTRUE p1.s
        MOV z5.s, #0
```

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

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

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

`CMPGT` (line 6), partitions the vector and sets the loop process-governing predicate `p0`. The lanes holding positive elements are set to active, while the remaining lanes are deactivated.

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

The input elements in register `z0` also act as the loop counter (in vector form). The next group of input elements is computed in a Vector Length Agnostic (VLA) approach, with instruction `DECW` (line 11). To obtain the input element of that lane for the next loop iteration, the current input element in each vector lane is decremented by the number of 32-bit elements in the implemented vector length. Then, with instruction `CMPGT` (line 12), the new vector partitions are based on the newly-calculated input elements. If at least one positive input element exists, the next loop iteration occurs. Otherwise, the loop exits. The conditional branch `B.FIRST` (line 13), checks if at least one positive input element exists.

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

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

```
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 = svcmpgt( p_all, svN_tmp, 0);

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

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

4.1.4 Effective vector length bandwidth utilization tips

Effective vectorization involves taking maximum advantage of the available load/store and processing bandwidth. The goal is to load and process enough elements to maximally fill up the available vector length in a Vector Length Agnostic (VLA) approach. To achieve a maximally filled vector length, often, in a vector register, you must arrange data in a specific way. Sometimes, to arrange the data in a specific way, you are also required to unroll the loop.

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

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

C reference:

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

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

Note

Filter coefficients are stored in memory in reverse order.

The SVE vectorized implementation:

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

        SXTW x1, w1
```

```
    ADD x1, x3, x1, LSL #1
    PTRUE p5.s

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

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

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

.L_return:
    RET
```

4.2 SVE Vector Length Agnostic (VLA) programming examples

This section provides several code examples that demonstrate Scalable Vector Extension (SVE) Vector Length Agnostic (VLA) programming.

The examples compute:

- The maximum element of a vector
- The dot-product of two vectors elements
- Finite Impulse Response (FIR) filtering
- Matrix multiplication

4.2.1 Vector maximum with real 16-bit integer 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 16-bit integer elements:

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

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

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

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

The following code example is the optimized SVE implementation:

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

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

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.

4.2.2 Vectors dot-product with complex SP floating-point elements and result

Describes an example SVE-optimized implementation of the dot-product computation of two vector elements with complex Single Precision (SP) floating-point input data and result.

Vector dot-product calculation

Vector dot-product is computed as:

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

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

Example

```
      size .req x0 // N
      aPtr .req x1 // complex float32_t * a
      bPtr .req x2 // complex float32_t * b
      outPtr .req x3 // complex float32_t * c

      DUP z8.d, #0
      DUP z9.d, #0
      PTRUE p2.s

      ADD size, aPtr, size, LSL #3
      INCB aPtr
      WHILELT p4.b, aPtr, size
      B.NFRST .L_tail_vecdot

    .L_unrolled_loop_vecdot:
      LD1W z0.s, p2/z, [aPtr, #-1, MUL VL]
      LD1W z1.s, p4/z, [aPtr]
```

```
18      LD1W z4.s, p2/z, [bPtr]
19      LD1W z5.s, p4/z, [bPtr, #1, MUL VL]
20
21      FCMLA z8.s, p2/m, z0.s, z4.s, #0 // c0 += a0*b0
22      FCMLA z8.s, p2/m, z0.s, z4.s, #90
23      FCMLA z9.s, p2/m, z1.s, z5.s, #0 // c1 += a1*b1
24      FCMLA z9.s, p2/m, z1.s, z5.s, #90
25
26      INCB aPtr, ALL, MUL #2
27      INCB bPtr, ALL, MUL #2
28      WHILELT p4.b, aPtr, size
29      B.FIRST .L_unrolled_loop_vecdot
30
31  .L_tail_vecdot:
32      DECB aPtr
33      WHILELT p4.b, aPtr, size
34      B.NFRST .L_return_vecdot
35
36      LD1W z3.s, p4/z, [aPtr]
37      LD1W z7.s, p4/z, [bPtr]
38
39      FCMLA z8.s, p4/m, z3.s, z7.s, #0
40      FCMLA z8.s, p4/m, z3.s, z7.s, #90
41
42  .L_return_vecdot:
43      UZP1 z10.s, z8.s, z9.s
44      UZP2 z11.s, z8.s, z9.s
45      FADDV s10, p2, z10.s
46      FADDV s11, p2, z11.s
47      STP s10, s11, [outPtr]
48
49      RET
```

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:

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

the first `FCMLA` instruction produces the following result:

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

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

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

Therefore:

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

After completing the processing of the vectorized loop and the termination code, the partial vector dot-product complex SP floating-point results are available in each 64-bit lane of the vector registers `z8` and `z9`, with the resulting real and imaginary parts interleaved.

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

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

4.2.3 FIR filtering with real SP floating-point elements

Describes an example SVE-optimized implementation of FIR filtering, with the real Single Precision (SP) floating-point input data, result, and FIR filter coefficients.

FIR filtering theory and C implementation

This topic illustrates the implementable FIR filtering theory.

The implementable FIR filtering is represented by:

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

The C code that implements the FIR filtering is:

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

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

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

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

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

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

Example

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.

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

4.2.4 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:

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

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

```
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:

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

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

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

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

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

The SVE vectorization is implemented by unrolling:

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

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

$$out[4, vl] = inLeft[4, K] * inRight[K, vl]$$

The computation is implemented using the floating-point multiply-add FMLA by indexed elements instruction (lines 46-56). The indexed elements are left-input matrix elements, which are loaded (two elements per one vector load) and replicated by instruction `LD1RQD` (lines 41-44). Right-input matrix elements are loaded by the contiguous vector length load `LD1D` (lines 38-39). The results of the innermost loop completion subblock, `out[4, vl]`, are stored to the memory by the contiguous vector length store `ST1D` (lines 62-65).

4.2.5 Matrix multiplication with real HP floating-point elements

The Higher Precision (HP) floating-point matrix multiplication:

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

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:

```
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:

```
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:

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

    float16_t *ptrIN;
    float16_t *ptrOUT;

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

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

     init_nb_elems = 8*K/vl;

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

        nb_st_elems = init_nb_elems;

        for (y=0; svptest_first( p16_all, p_ld = svwhilelt_b16(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]);
39              r6 = svld1(p_ld, &ptrIN[offsetIN_6]);
40              r7 = svld1(p_ld, &ptrIN[offsetIN_7]);
41
42              svfloat16_t t8 = svzip1(r0, r4);
43              svfloat16_t t9 = svzip1(r2, r6);
44              svfloat16_t t10 = svzip1(r1, r5);
45              svfloat16_t t11 = svzip1(r3, r7);
46              svfloat16_t t12 = svzip2(r0, r4);
47              svfloat16_t t13 = svzip2(r2, r6);
48              svfloat16_t t14 = svzip2(r1, r5);
49              svfloat16_t t15 = svzip2(r3, r7);
50
51              svfloat16_t t16 = svzip1(t8, t9);
52              svfloat16_t t17 = svzip1(t10, t11);
53              svfloat16_t t18 = svzip2(t8, t9);
54              svfloat16_t t19 = svzip2(t10, t11);
55              r0 = svzip1(t16, t17);
56              r1 = svzip2(t16, t17);
57              r2 = svzip1(t18, t19);
58              r3 = svzip2(t18, t19);
59
60              t16 = svzip1(t12, t13);
61              t17 = svzip1(t14, t15);
62              t18 = svzip2(t12, t13);
63              t19 = svzip2(t14, t15);
64              r4 = svzip1(t16, t17);
65              r5 = svzip2(t16, t17);
66              r6 = svzip1(t18, t19);
67              r7 = svzip2(t18, t19);
68
69              switch(nb_st_elems) {
70                  default :
71                      svst1_vnum(p16_all, ptrOUT, 7, r7);
72                      svst1_vnum(p16_all, ptrOUT, 6, r6);
73                      svst1_vnum(p16_all, ptrOUT, 5, r5);
74                      svst1_vnum(p16_all, ptrOUT, 4, r4);
75                  case 4 :
76                      svst1_vnum(p16_all, ptrOUT, 3, r3);
77                      svst1_vnum(p16_all, ptrOUT, 2, r2);
78                  case 2 :
79                      svst1_vnum(p16_all, ptrOUT, 1, r1);
80                      svst1(p16_all, ptrOUT, r0);
81              }
82
83              ptrIN += vl;
84              ptrOUT += 8*vl;
85              nb_st_elems -= 8;
86          }
87      }
88  }
```

The transpose of the 8 matrix rows is implemented using the `ZIP1` and `ZIP2` instructions, in 3 steps (lines 42-67). Because the matrix dimensions are multiples of 16, and element size is 16-bit, 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:

```
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 = svptrue_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 = svmla_lane(acc0, inR, inL, 0);
                acc1 = svmla_lane(acc1, inR, inL, 1);
                acc2 = svmla_lane(acc2, inR, inL, 2);
                acc3 = svmla_lane(acc3, inR, inL, 3);
                acc4 = svmla_lane(acc4, inR, inL, 4);
                acc5 = svmla_lane(acc5, inR, inL, 5);
                acc6 = svmla_lane(acc6, inR, inL, 6);
                acc7 = svmla_lane(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:

$$out[8, vl] = inLeft[8, K] * inRight[K, vl]$$

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

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_ld_st`) set, at line 28.

4.2.6 Matrix multiplication with real 8-bit integer input elements and real 32-bit integer output elements

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

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

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

is implemented in two steps. In the first step, both 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:

```
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, vl/4], where vl 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:

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

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

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

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

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

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

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

The following C code shows the calculations:

```
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 matrix multiplication with real 8-bit integer input elements and real 32-bit integer 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:

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

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

     uint8_t *ptrIN;
     uint8_t *ptrOUT;

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

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

     init_nb_elems = 8*K/vl;

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

        nb_st_elems = init_nb_elems;

        for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, K)); y+=vl) {
            r0 = svld1(p_ld, ptrIN);
            r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
            r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
            r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
            r4 = svld1(p_ld, &ptrIN[offsetIN_4]);
40              r5 = svld1(p_ld, &ptrIN[offsetIN_5]);
41              r6 = svld1(p_ld, &ptrIN[offsetIN_6]);
42              r7 = svld1(p_ld, &ptrIN[offsetIN_7]);
43
44              t8 = svzip1(svreinterpret_u32(r0), svreinterpret_u32(r4));
45              t9 = svzip1(svreinterpret_u32(r2), svreinterpret_u32(r6));
46              t10 = svzip1(svreinterpret_u32(r1), svreinterpret_u32(r5));
47              t11 = svzip1(svreinterpret_u32(r3), svreinterpret_u32(r7));
48              t12 = svzip2(svreinterpret_u32(r0), svreinterpret_u32(r4));
49              t13 = svzip2(svreinterpret_u32(r2), svreinterpret_u32(r6));
50              t14 = svzip2(svreinterpret_u32(r1), svreinterpret_u32(r5));
51              t15 = svzip2(svreinterpret_u32(r3), svreinterpret_u32(r7));
52
53              t16 = svzip1(t8, t9);
54              t17 = svzip1(t10, t11);
55              t18 = svzip2(t8, t9);
56              t19 = svzip2(t10, t11);
57              r00 = svzip1(t16, t17);
58              r11 = svzip2(t16, t17);
59              r22 = svzip1(t18, t19);
60              r33 = svzip2(t18, t19);
61
62              t16 = svzip1(t12, t13);
63              t17 = svzip1(t14, t15);
64              t18 = svzip2(t12, t13);
65              t19 = svzip2(t14, t15);
66              r44 = svzip1(t16, t17);
67              r55 = svzip2(t16, t17);
68              r66 = svzip1(t18, t19);
69              r77 = svzip2(t18, t19);
70
71              switch(nb_st_elems) {
72                  default :
73                      svst1_vnum(p8_all, ptrOUT, 7, svreinterpret_u8(r77));
74                      svst1_vnum(p8_all, ptrOUT, 6, svreinterpret_u8(r66));
75                      svst1_vnum(p8_all, ptrOUT, 5, svreinterpret_u8(r55));
76                      svst1_vnum(p8_all, ptrOUT, 4, svreinterpret_u8(r44));
77                  case 4 :
78                      svst1_vnum(p8_all, ptrOUT, 3, svreinterpret_u8(r33));
79                      svst1_vnum(p8_all, ptrOUT, 2, svreinterpret_u8(r22));
80                  case 2 :
81                      svst1_vnum(p8_all, ptrOUT, 1, svreinterpret_u8(r11));
82                      svst1(p8_all, ptrOUT, svreinterpret_u8(r00));
83              }
84
85              ptrIN += vl;
86              ptrOUT += 8*vl;
87              nb_st_elems -= 8;
88          }
89      }
90  }
```

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

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:

```
void rearrangeRight_fixp( uint64_t K, uint64_t N,
2                             uint8_t * inRight, uint8_t * inRight_MOD) {
3       uint64_t x, y, nb_st_elems;
4       svbool_t p_ld;
5       svuint8_t r0, r1, r2, r3;
6       uint64_t offsetIN_1, offsetIN_2, offsetIN_3;
7       uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3;
8
9       uint8_t *ptrIN;
10      uint8_t *ptrOUT;
11
12      svbool_t p8_all = svptrue_b8();
13      uint64_t vl = svcntb();
14      uint64_t vl_4 = (vl >> 2);
15
16      offsetIN_1 = N;
17      offsetIN_2 = 2*N;
18      offsetIN_3 = 3*N;
19
20      offsetOUT_1 = K*vl_4;
21      offsetOUT_2 = K*vl_4*2;
22      offsetOUT_3 = K*vl_4*3;
23
24      nb_st_elems = 4*N/vl;
25
26      for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, N)); y+=vl) {
27          ptrOUT = &inRight_MOD[y*K];
28
29          for (x=0; x<K; x+=4) {
30              ptrIN = &inRight[x*N+y];
31
32              r0 = svld1(p_ld, ptrIN);
33              r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
34              r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
35              r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
36
37              svuint8_t t4 = svzip1(r0, r2);
38              svuint8_t t5 = svzip1(r1, r3);
39              svuint8_t t6 = svzip2(r0, r2);
40              svuint8_t t7 = svzip2(r1, r3);
41
42              r0 = svzip1(t4, t5);
43              r1 = svzip2(t4, t5);
44              r2 = svzip1(t6, t7);
45              r3 = svzip2(t6, t7);
46
47              switch(nb_st_elems) {
48                  default :
49                      svst1(p8_all, &ptrOUT[offsetOUT_3], r3);
50                      svst1(p8_all, &ptrOUT[offsetOUT_2], r2);
51                  case 2 :
52                      svst1(p8_all, &ptrOUT[offsetOUT_1], r1);
53                  case 1 :
54                      svst1(p8_all, ptrOUT, r0);
55              }
56
57              ptrOUT += vl;
58          }
59
60          nb_st_elems -= 4;
61      }
62  }
```

The rearrangement of 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:

```
1   void matmul_dotp_fixp( uint64_t M, uint64_t K, uint64_t N,
2                          uint8_t * inLeft_MOD, uint8_t * inRight_MOD, uint32_t *
 out) {
3       uint64_t x, y, z;
4       svuint32_t acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
5       uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3, offsetOUT_4;
6       uint64_t offsetOUT_5, offsetOUT_6, offsetOUT_7;
7
8       uint8_t *ptrIN_left;
9       uint8_t *ptrIN_right;
10      uint32_t *ptrOUT;
11
12      svbool_t p8_all = svptrue_b8();
13      uint64_t vl = svcntb();
14      uint64_t vl_4 = (vl >> 2);
15
16      offsetOUT_1 = N;
17      offsetOUT_2 = 2*N;
18      offsetOUT_3 = 3*N;
19      offsetOUT_4 = 4*N;
20      offsetOUT_5 = 5*N;
21      offsetOUT_6 = 6*N;
22      offsetOUT_7 = 7*N;
23
24      for (x=0; x<M; x+=8) {
25          ptrOUT = &out[x*N];
26          ptrIN_right = &inRight_MOD[0];
27
28          for (y=0; y<N; y+=vl_4) {
29              ptrIN_left = &inLeft_MOD[x*K];
30
31              acc0 = svdup_u32(0);
32              acc1 = svdup_u32(0);
33              acc2 = svdup_u32(0);
34              acc3 = svdup_u32(0);
35              acc4 = svdup_u32(0);
36              acc5 = svdup_u32(0);
37              acc6 = svdup_u32(0);
38              acc7 = svdup_u32(0);
39
40              for (z=0; z<K; z+=4) {
41                  svuint8_t a0123 = svld1rq(p8_all, ptrIN_left);
42                  svuint8_t a4567 = svld1rq(p8_all, &ptrIN_left[16]);
43                  svuint8_t b_vec = svld1(p8_all, ptrIN_right);
44
45                  acc0 = svdot_lane(acc0, b_vec, a0123, 0);
46                  acc1 = svdot_lane(acc1, b_vec, a0123, 1);
47                  acc2 = svdot_lane(acc2, b_vec, a0123, 2);
48                  acc3 = svdot_lane(acc3, b_vec, a0123, 3);
49                  acc4 = svdot_lane(acc4, b_vec, a4567, 0);
50                  acc5 = svdot_lane(acc5, b_vec, a4567, 1);
51                  acc6 = svdot_lane(acc6, b_vec, a4567, 2);
52                  acc7 = svdot_lane(acc7, b_vec, a4567, 3);
53
54                  ptrIN_left += 32;
55                  ptrIN_right += vl;
56              }
57
58              svst1(p8_all, ptrOUT, acc0);
59              svst1(p8_all, &ptrOUT[offsetOUT_1], acc1);
60                svst1(p8_all, &ptrOUT[offsetOUT_2], acc2);
61                svst1(p8_all, &ptrOUT[offsetOUT_3], acc3);
62                svst1(p8_all, &ptrOUT[offsetOUT_4], acc4);
63                svst1(p8_all, &ptrOUT[offsetOUT_5], acc5);
64                svst1(p8_all, &ptrOUT[offsetOUT_6], acc6);
65                svst1(p8_all, &ptrOUT[offsetOUT_7], acc7);
66
67                ptrOUT += vl_4;
68            }
69        }
70    }
```

The SVE vectorization is implemented by unrolling:

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

The innermost loop, at line 40, produces results of a subblock:

$$out[8, vl = 4] = inLeft[8, K] * inRight[K, vl = 4]$$

The computation is implemented using the unsigned integer dot-product, by indexed elements `UDOT` instruction, at lines 45-52. The indexed elements are left-input matrix elements that are loaded and replicated by instruction `LD1RQB` (16 elements per one vector load) at lines 41-42. Right-input matrix elements are loaded by contiguous vector length load `LD1B` at line 43. At completion of the innermost loop a subblock:

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

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.

4.3 SVE Vector Length Specific (VLS) programming

This chapter introduces the concept of Vector Length Specific (VLS) programming and provides some Scalar Vector Extension (SVE) programming tips. To support the tips, assembly examples that use the Arm C Language Extensions (ACLE) for SVE are provided.

SVE is an architecture extension that supports a range of different vector register sizes. Some SVE ACLE code can be compatible with all of these register sizes; such code is considered Vector Length Agnostic (VLA). Other SVE ACLE code requires the register size to belong to a restricted set; such code is considered Vector Length Specific (VLS). Restrictions can include imposing a minimum register size, a maximum register size, a power-of-two register size, or a single exact register size.

When you implement your code, you can choose to use fixed-length vectors. Fixed-length vectors enable the use of constructs that are generally not safe for code which is to be run on targets with unknown SVE vector lengths. However, if you do not require your code to be portable, VLS code can be more optimal than VLA code for a specific SVE implementation.

Arm® Compiler for Linux 23.04 does not support autovectorization with VLS programming. To compile VLS code with Arm Compiler for Linux 23.04, you must write code that uses ACLE for SVE.

In the program code, the vector length can be controlled using the `arm_sve_vector_bits` ACLE attribute. At compile time, set the vector length using the `-msve-vector-bits=<arg>` compiler option.

Generated VLS code must only be executed on hardware which offers an SVE vector length compatible with the intent of the programmer. To check the SVE vector length of the target at compile-time, use the `__ARM_FEATURE_SVE_BITS` define, as described in detail in Arm C Language Extensions for SVE specification.

4.3.1 SVE VLS code example

Describes how to protect SVE code to be Vector Length Specific (VLS).

The following example shows how to protect Arm C Language Extension (ACLE) code at compile time, for an intended SVE vector length of 512 bits:

```
#include <arm_sve.h>
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vls_vec_t __attribute__((arm_sve_vector_bits(512)));
#else
#error Only -msve-vector-bits=512 is supported
#endif

// Function to add 2 input vectors.
vls_vec_t vls_add(vls_vec_t a, vls_vec_t b) {
    return svadd_s32_x(svptrue_b32(), a, b);
}
```

To specify the SVE vector length, the vector types are declared with the `arm_sve_vector_bits` attribute.

To compile the example, on the compile line, include the `-msve-vector-bits=<arg>` option and replace `<arg>` with the SVE vector length. For example:

```
armclang -march=armv8-a+sve -O2 -msve-vector-bits=512 -c c-acle-sve-vector-bits-
simple.c
```

Note

If you do not specify the `-msve-vector-bits=<arg>` option, a compile-time error occurs, through the `#error` directive.

Arm® Compiler for Linux 23.04 generates:

```
vls_add:
    add     z0.s, z0.s, z1.s
    ret
```

which uses the SVE instruction set.

4.3.2 Choose the vector length

This example shows why some Scalable Vector Extension (SVE) Arm C Language Extensions (ACLE) code requires the SVE vector length to be restricted.

You can code the following function using vector operations:

```
void vls_slp(double a1, double a2, double a3, double a4,
         double b1, double b2, double b3, double b4,
         double *A) {
  A[0] = a1*(a1 + b1)/b1;
  A[1] = a2*(a2 + b2)/b2;
  A[2] = a3*(a3 + b3)/b3;
  A[3] = a4*(a4 + b4)/b4;
}
```

The sequence of operations is the same in every line of code. If the input variables are gathered in vectors (`a` and `b`), you can write the code to perform all four lines of code in parallel with vector operations. The `vls_slp` function computes four variables of double type, which makes the function suitable for an SVE vector length of 256 bits. You could also write `vls_slp` with the following SVE VLS ACLE code:

```
#include <arm_sve.h>
#if __ARM_FEATURE_SVE_BITS==256
typedef svfloat64_t vec_float_t __attribute__((arm_sve_vector_bits(256)));
#else
#error Only -msve-vector-bits=256 is supported
#endif

void vls_slp(double a1, double a2, double a3, double a4,
          double b1, double b2, double b3, double b4,
          double *A) {
    svbool_t pg = svptrue_b64();
    double a[4] = {a1, a2, a3, a4};
    vec_float_t vec_a = svld1_f64(pg, &a[0]);
    double b[4] = {b1, b2, b3, b4};
    vec_float_t vec_b = svld1_f64(pg, &b[0]);
    vec_float_t add_res = svadd_f64_z(pg, vec_a, vec_b);
    vec_float_t mul_res = svmul_f64_z(pg, vec_a, add_res);
    vec_float_t div_res = svdiv_f64_z(pg, mul_res, vec_b);
    svst1_f64(pg, A, div_res);
}
```

To compile the example (`c-acle-sve-vector-bits-choosing-vl.c`), on the compile line, include the `-msve-vector-bits=<arg>` option and specify 256 bits. For example:

```
 armclang -march=armv8-a+sve -c -O2 -msve-vector-bits=256 c-acle-sve-vector-bits-
choosing-vl.c
```

Arm® Compiler for Linux 23.04 generates:

```
vls_slp:
    sub sp, sp, #0x40
    stp d0, d1, [sp, #32]
    stp d2, d3, [sp, #48]
    ptrue p0.d
    add x8, sp, #0x20
    ld1d {z0.d}, p0/z, [x8]
    stp d4, d5, [sp]
    stp d6, d7, [sp, #16]
    mov x8, sp
    ld1d {z1.d}, p0/z, [x8]
    movprfx z2, z0
    fadd z2.d, p0/m, z2.d, z1.d
    fmul z0.d, p0/m, z0.d, z2.d
    fdiv z0.d, p0/m, z0.d, z1.d
    st1d {z0.d}, p0, [x0]
    add sp, sp, #0x40
    ret
```

Note

The generated code will not produce the intended result if the hardware SVE vector length is not 256 bits. If the vector length is shorter, the vector code does not cover all the lines of code from the original function. If the vector length is longer, then the store might write to unallocated memory. If required, you can modify SVE ACLE code to take the new vector length into account. You must recompile modified code with Arm Compiler for Linux, and use `-msve-vector-bits=<arg>` to set the new SVE vector width.

4.4 Migrate from Neon to SVE

This guide summarizes the important differences between coding for the Scalable Vector Extension (SVE) and coding for Neon®. For users who have already ported their applications to Arm®v8-A Neon hardware, the guide also highlights the key differences to consider when porting an application to SVE.

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

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

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

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

4.4.1 Migrate from Neon to SVE: Before you begin

References the resources to read before reading this tutorial.

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

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

4.4.2 Part 1: Neon and SVE fundamentals

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

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

Instruction sets

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

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

Table 4-2: Summary of the Instruction Set extensions

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

Registers, vectors, lanes, and elements

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

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

All elements in a vector have the same data type.

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

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

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

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

- The `Z` registers are data registers. `Z` register bits are an IMPLEMENTATION DEFINED multiple of 128, up to an architectural maximum of up to 2048-bits. Data in these registers can be interpreted as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements. The low 128 bits of each `Z` register overlap the corresponding Neon registers, and therefore also the scalar floating-point registers.
- The `P` registers hold one bit for each byte available in a `Z` register. In other words, a `P` register is always 1/8th the size of a `Z` register. Predicated instructions use a P register to determine

which vector elements to process. Each individual bit in the `P` register specifies whether the corresponding byte in the `Z` register is active or inactive.

- The FFR register is a dedicated predicate register that that captures the cumulative fault status of a sequence of SVE vector load instructions. SVE provides a first-fault option for some SVE vector load instructions. This option suppresses memory access faults if they do not occur as a result of the first active element of the vector. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded.

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

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

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

Compiling an application without knowing the vectorized length means:

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

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

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

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

Vector Length Specific (VLS) programming

Vector Length Specific (VLS) code is code that has been written for a specific set of restricted register sizes. Restrictions can include a minimum register size, a maximum register size, a power-of-two register size, or a single exact register size.

While a key goal for SVE is to support VLA programming, and the convenience of portable VLA SVE code, you can still write VLS SVE code. VLS SVE code is code that has been written for a specific SVE vector length. VLS SVE code is not portable across SVE systems which have different SVE vector lengths implemented. However, if you know the specific vector length of your SVE-based target, and you do not require your code to be portable to SVE targets with a different or unknown vector length, you can create a highly-optimized binary using VLS SVE code.

Arm Compiler for Linux 23.04 does not support autovectorization with VLS programming. To compile VLS code with Arm Compiler for Linux 23.04, you must write code that uses ACLE for SVE. In your program code, the vector length can be controlled using the `arm_sve_vector_bits` ACLE attribute. At compile time, set the vector length using the `-msve-vector-bits=<arg>` compiler option.

To learn more about VLS SVE programming, and see some VLS SVE code examples, see SVE Vector Length Specific (VLS) programming.

4.4.3 Part 2: Preparing to migrate your optimized Neon code to SVE

As a programmer, there are various ways you can use Neon and Scalable Vector Extension (SVE) technology.

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

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

- **Compiler auto-vectorization**

  Auto-vectorization features in your compiler allow the compiler to automatically optimize SVE code.
- **Intrinsics**

  Intrinsics are function calls that the compiler replaces with appropriate instructions. SVE intrinsics are available and give you direct access to the exact instructions you want.
- **Libraries**

  Math libraries are a set of optimized math routines for a particular architecture or target. Math libraries are available in SVE variants, containing optimized SVE routines, respectively. For example Arm Performance Libraries.
- **Assembly**

  To fine tune your code and have the highest possible control over performance, experienced programmers can code in SVE assembly. If you are comfortable coding in assembly, you can use the SVE specification to find out what instructions are available for SVE code, and use them to write your application assembly.

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

Compiler auto-vectorization

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

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

Note

Vector Length Specific (VLS) SVE autovectorization is not supported in Arm Compiler for Linux 23.04.

If you enable autovectorization in Arm Compiler for Linux 23.04, the compiler generates a Vector Length Agnostic (VLA) binary.

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

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

The benefits of relying on compiler auto-vectorization are:

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

To enable the compiler to auto-vectorize your code:

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

Auto-vectorization compiler options

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

Table 4-3: Arm Compiler for Linux auto-vectorization optimization options

Option	Description	Auto-vectorization
`-O0`	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.	Never

Option	Description	Auto-vectorization
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.	Disabled by default.
`-O2`	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.	Enabled by default.
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.	Enabled by default.
`-Ofast`	Enable all the optimizations from level 3, including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.	Enabled by default.

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

Note

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

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

Use the restrict keyword

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

Provide hints to the compiler

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

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

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

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

Note

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

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

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

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

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

Header files and processor optimization

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

Table 4-4: Auto-vectorization with Arm Compiler for Linux

<table>
<tr><th>Extension</th><th>Header file form</th><th>Recommended compilation options</th><th>Notes</th></tr>
<tr><td>Neon</td><td><code>#include <arm_neon.h></code></td><td><code>armclang -O<2|3|fast> -mcpu={native|<target>} -o <binary_name> <filename>.c</code></td><td><code>-mcpu</code> enables the compiler to use micro-architectural optimizations, and can be set to a specific target (<code><target></code>), or you can allow the compiler to determine what processor it is running on (<code>native</code>).</td></tr>
<tr><td>SVE</td><td><code>#ifdef</code>
<code> __ARM_FEATURE_SVE</code>
<code>#include <arm_sve.h></code>
<code>#endif /*</code>
<code> __ARM_FEATURE_SVE */</code></td><td>To run on SVE-enabled hardware:
<code>armclang -O<2|3|fast> -mcpu={native|<target>} -o <binary_name> <filename>.c</code>
To produce SVE code for emulation on hardware that is not SVE-enabled:
<code>armclang -O<2|3|fast> -march=armv8-a+sve -o <binary_name> <filename>.c</code></td><td>Like for Neon, <code>-mcpu</code> enables the compiler to use micro-architectural optimizations. If the target is SVE-enabled, the compiler will produce optimized SVE code, rather than optimized Neon code.
As a less-optimal alternative to <code>-mcpu</code>, <code>-march=armv8-a+sve</code> tells the compiler to optimize for SVE-enabled Armv8-A hardware, but without the microarchitectural optimizations. SVE code is produced and you can use Arm Instruction Emulator (ArmIE) to emulate the SVE instructions on any Armv8-A hardware..
<b>Note:</b>
When SVE-enabled hardware is available and you are compiling on that target SVE hardware, Arm recommends using <code>-mcpu=native</code> so that micro-architectural optimizations can be taken advantage of.
To read more about the <code>-march</code> and <code>-mcpu</code> options, as well as <code>-mtune</code>, see the Compiler flags across architectures: -march, -mtune, and -mcpu blog.</td></tr>
</table>

Intrinsics

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

Using intrinsics has several benefits:

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

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

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

Program conventions: macros, types, and functions

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

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

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

Macros

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

Note

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

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

Table 4-5: Neon macros

Macro	Description
`__aarch64__`	Selection of architecture-dependent source at compile time.

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

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

Table 4-6: SVE macros

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

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

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

Data types

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

Data types - Neon

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

Table 4-7: Neon data types

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

Where:

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

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

Data types - SVE

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

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

Table 4-8: SVE data types

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

Where:

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

If you are writing VLS SVE ACLE code, you must define fixed versions of the SVE types which you can pass to the SVE intrinsics in your code. Your SVE intrinsics can then depend on the value of `__ARM_FEATURE_SVE_BITS`. For example, a VLS SVE ACLE example that depends on a vector width of 512 bits could be:

```
#include <arm_sve.h>
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vls_vec_t __attribute__((arm_sve_vector_bits(512)));
#else
#error Only -msve-vector-bits=512 is supported
#endif

// Function to add 2 input vectors.
vls_vec_t vls_add(vls_vec_t a, vls_vec_t b) {
    return svadd_s32_x(svptrue_b32(), a, b);
}
```

For more information, see SVE Vector Length Specific (VLS) programming and the Arm C/C++ Compiler Developer and Reference Guide.

Functions

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

Functions - Neon

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

At the most general level, this is:

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

For example:

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

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

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

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

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

Table 4-9: Neon functions pattern

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

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

Functions - SVE

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

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

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

The following table describes the different pattern elements:

Table 4-10: SVE functions pattern

Pattern element	Description
`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. In many cases, these are optional.
`_predication`	This suffix describes the inactive elements in the result of a predicated operation. It can be one of `z` (zero predication), `m` (merge predication), or `x` ('Do not care' predication).

For more information, see the ACLE for SVE specification.

Intrinsics resources

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

The following intrinsics resources are also available:

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

Libraries

Arm Performance Libraries contains optimized numerical libraries for both Neon and SVE-enabled processors.

When linking against Arm Performance Libraries, use the `-armpl` Arm Compiler for Linux options, (or `-larmpl[_lp64|_ilp64]{_mp}]` for GCC compilers).

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

Assembly

If you are familiar with writing Neon assembly, you can use the Arm C Language Extensions (ACLE) for SVE specification to find out about the instructions SVE provides, and the Procedure Call

Standard (PCS) with SVE support to learn about SVE register assignment. Once you understand these documents, you can analyze your Neon assembly and update it to use SVE instructions, where possible.

Some useful tips to improve performance in SVE assembly:

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

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

Note

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

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

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

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

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

4.4.4 Part 3: When it is sometimes useful to keep optimized Neon code

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

1. Sparse predication overhead:

   Sometimes, the cost of using complete scalar Neon register can be lower than computing a partial vector using a predicated full-length SVE register.

   For example, where `inner` is <= 0.5*VL in:

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

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

2. General VLA overhead:

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

   For example, in:

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

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

4.4.5 Part 4: Migrate your Neon code to SVE

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

- To migrate to VLA or VLS SVE code in a high-level language like C/C++ or Fortran, you should:
  1. Update your compiler options (including auto-vectorization options).

2. Where possible, and by referring to the SVE specification, replace or re-write your code to use SVE *intrinsics* instead of Neon® *intrinsics*.
3. For VLS code, compile your code specifically for the vector width you intend. Control the vector length using the `arm_sve_vector_bits` Arm C Languages Extensions (ACLE) attribute, and when using Arm® Compiler for Linux, use the `-msve-vector-bits=<arg>` option on your compile command line.
4. Link your code with the SVE variants of the math libraries.

- If you are writing VLA or VLS assembly, you should:
  1. Update your compiler options (including auto-vectorization options).
  2. Where possible, and by referring to the SVE specification, re-write your code to use SVE *instructions* instead of Neon *instructions*.
  3. For VLS code, compile your code specifically for the vector width you intend. When using Arm Compiler for Linux, use the `-msve-vector-bits=<arg>` option on your compile command line.
  4. Link your code with the SVE variants of the math libraries.

Note

Arm Compiler for Linux 23.04 does not support VLS autovectorization.

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

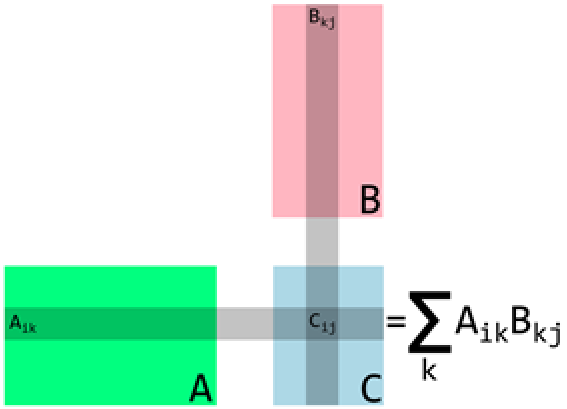
The examples do not cover re-writing Neon optimized assembly.

Example 1: Rewriting a simple matrix multiplication code using intrinsics

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

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

Figure 4-1: Matrix multiplication diagram



The matrix multiplication process is as follows:

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

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

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

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

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

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

```
void matrix_multiply_4x4_neon(const float32_t *A, const float32_t *B, float32_t *C)
 {
    // these are the columns A
    float32x4_t A0;
    float32x4_t A1;
    float32x4_t A2;
    float32x4_t A3;
    // these are the columns B
    float32x4_t B0;
    float32x4_t B1;
    float32x4_t B2;
    float32x4_t B3;

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

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

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

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

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

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

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

Fixed-size 4x4 matrices are chosen because:

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

The Neon intrinsics that are used in this example are:

Table 4-11: Summary of Neon intrinsics used

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

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

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

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

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

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

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

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

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

    B2 = svld1rq_f32(svptrue_b32(), B+8);
    C2 = svmla_lane_f32(C2, A0, B2, 0);
    C2 = svmla_lane_f32(C2, A1, B2, 1);
    C2 = svmla_lane_f32(C2, A2, B2, 2);
    C2 = svmla_lane_f32(C2, A3, B2, 3);
    svst1_f32(pred, C+8, C2);

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

The SVE intrinsics that are used in this example are:

Table 4-12: Summary of SVE intrinsics used

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

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

Example 2: Rewriting a larger matrix multiplication code with intrinsics

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

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

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

    int a_idx;
    int b_idx;
    int c_idx;

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

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

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

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

                // load most current a values in row
                A0 = vld1q_f32(A+a_idx);
                A1 = vld1q_f32(A+a_idx+n);
                A2 = vld1q_f32(A+a_idx+2*n);
                A3 = vld1q_f32(A+a_idx+3*n);
                // multiply accumulate 4x1 blocks, that is each column C
                B0 = vld1q_f32(B+b_idx);
                C0 = vfmaq_laneq_f32(C0,A0,B0,0);
                C0 = vfmaq_laneq_f32(C0,A1,B0,1);
                C0 = vfmaq_laneq_f32(C0,A2,B0,2);
                C0 = vfmaq_laneq_f32(C0,A3,B0,3);

                B1 = v1d1q_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,B3,3);

                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.
- `FMLA` instead of `FMUL` instructions. As specified by the intrinsics.
- Fewer loop iterations. When used properly intrinsics allow loops to be unrolled easily.

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

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

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

    int a_idx;
    int b_idx;
    int c_idx;

    // these are the columns of a nx4 sub matrix of A
    svfloat32_t A0;
    svfloat32_t A1;
    svfloat32_t A2;
    svfloat32_t A3;
    // these are the columns of a 4x4 sub matrix of B
    svfloat32_t B0;
    svfloat32_t B1;
    svfloat32_t B2;
    svfloat32_t B3;

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

    for (int i_idx=0; i_idx<n; i_idx+=svcntw()) {
        // calculate predicate for this i_idx
        svbool_t pred = svwhilelt_b32_u32(i_idx, n);

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

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

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

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

                B2 = svld1rq_f32(svptrue_b32(), B+b_idx+2*k);
                C2 = svmla_lane_f32(C2,A0,B2,0);
                C2 = svmla_lane_f32(C2,A1,B2,1);
                C2 = svmla_lane_f32(C2,A2,B2,2);
                C2 = svmla_lane_f32(C2,A3,B3,3);

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

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

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

- Uses `WHILELT` to determine the predicate for doing each iteration of the outer loop. This guarantees you have at least one element to do by the loop condition.
- Increments `i_idx` by `CNTW` (the number of 32-bit elements in a vector) to avoid hard-coding the number of elements calculated in an iteration of the outer loop.

4.4.6 Check your knowledge

Read the following questions to check your knowledge.

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

  `arm_neon.h` (Neon®) and `arm_sve.h` (SVE)
- **What size are the Neon and SVE vector data registers?**

  Neon registers are 128-bit registers. SVE does not define a fixed size for its vector registers. SVE vector registers are an IMPLEMENTATION DEFINED multiple of 128 bits, up to an architectural maximum of up to 2048 bits.
- **What term describes allowing the compiler to automatically identify opportunities in your code to use Neon or SVE instructions?**

  Auto-vectorization

4.4.7 Related information

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

Related information

Arm C Language Extensions (ACLE)
Neon Intrinsics Reference
Architecture Exploration Tools
Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile
Arm C Language Extensions for SVE specification
Arm Server and HPC tools for SVE
ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for Armv8-A

4.5 Coding for SVE resources

This topic describes the range of learning resources that are available to you to learn more about the Arm Scalable Vector Extension (SVE).

- SVE Programmer's Guide

  Learn more about the SVE through a series of guides. From the fundamentals to more advanced concepts, these guides provide an introduction to the SVE extension to the Arm Armv8-A architecture.
- Past presentations and hackathon materials

  Past presentations at Arm events, including downloadable SVE Hackathon materials.
- Arm C Language Extensions (ACLE) for SVE

  The ACLE for SVE 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 Arm®v8-A architecture profile.