ARM® Compiler

Version 6.8

Scalable Vector Extension User Guide

ARM 100891_0608_00_en

ARM® Compiler

Scalable Vector Extension User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
C	04 November 2016	Non-Confidential	ARM Compiler v6.6 Release
0607-00	05 April 2017	Non-Confidential	ARM Compiler v6.7 Release. Document numbering scheme has changed.
0607-01	19 April 2017	Non-Confidential	ARM Update 1 for ARM Compiler 6.7. Added links to the ARM C Language Extensions for SVE specification on developer.arm.com.
0608-00	30 July 2017	Non-Confidential	ARM Compiler v6.8 Release. Removal of limitation to inclusion of arm_sve.h.

Non-Confidential Proprietary Notice

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents
ARM® Compiler Scalable Vector Extension User Guide

Chapter 5 Troubleshooting

List of Tables
ARM® Compiler Scalable Vector Extension User Guide

Preface

This preface introduces the *ARM® Compiler Scalable Vector Extension User Guide*.

It contains the following:

About this book

The ARM® Compiler Scalable Vector Extension User Guide provides information about using ARM Compiler 6 with targets that implement the Scalable Vector Extension (SVE) for ARMv8-A AArch64.

Using this book

This book is organized into the following chapters:

Chapter 1 SVE Overview

Gives general information about ARM Compiler 6 support for the Scalable Vector Extension (SVE) EAC (00rel1) for ARMv8-A AArch64.

Chapter 2 Getting Started with the SVE Compiler

Describes how to generate an executable binary that makes use of the instructions provided by the SVE architectural extension to the ARMv8-A architecture.

Chapter 3 Coding Considerations

Describes best practices for writing code that can encourage the SVE Compiler to produce optimal auto-vectorized output.

Chapter 4 Reference

Provides reference information about ARM Compiler.

Chapter 5 Troubleshooting

Provides general troubleshooting advice relating to SVE functionality in this release of ARM Compiler.

Glossary

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

See the *ARM® Glossary* for more information.

Typographic conventions

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

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

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

`monospace`
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

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

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

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

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

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

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

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

- The title *ARM Compiler Scalable Vector Extension User Guide*.
- The number ARM 100891_0608_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

Other information

- *ARM Developer*.
- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
SVE Overview

Gives general information about ARM Compiler 6 support for the Scalable Vector Extension (SVE) EAC (00rel1) for ARMv8-A AArch64.

It contains the following sections:

1.1 Introducing SVE

The ARM Compiler toolchain supports targets that implement the Scalable Vector Extension (SVE) EAC (00rel2) for ARMv8-A AArch64.

SVE is the next-generation SIMD instruction set for AArch64, that introduces the following new architectural features for *High Performance Computing* (HPC):

- Scalable vector length.
- Per-lane predication.
- Gather-load and scatter-store.
- Fault-tolerant speculative vectorization.
- Horizontal and serialized vector operations.

This release of the ARM Compiler toolchain lets you:

- Assemble source code containing SVE instructions.
- Disassemble ELF object files containing SVE instructions.
- Compile C and C++ code for SVE-enabled targets, with an advanced auto-vectorizer capable of taking advantage of SVE features.
- Use intrinsics to write SVE instructions directly from C code.

Note

The ARM Compiler toolchain only supports bare-metal applications.

This document provides information about the features of the ARM Compiler toolchain that specifically relate to SVE. For information about the other features, see the ARM Compiler documentation.

Related information

ARM Compiler 6 documentation.

1.2 License configuration for SVE

Use of the SVE functionality in the ARM Compiler toolchain requires a suitable valid license. Contact your ARM support representative for more information.

To indicate the location of your license, set the `ARMLMD_LICENSE_FILE` environment variable to reference either your license file or license server:

```
export ARMLMD_LICENSE_FILE=/home/user/sve.lic
export ARMLMD_LICENSE_FILE=27000@license-server-hostname
```

Related information

ARM DS-5 License Management Guide.

ARM Compiler User Guide: Toolchain environment variables.

Chapter 2
Getting Started with the SVE Compiler

Describes how to generate an executable binary that makes use of the instructions provided by the SVE architectural extension to the ARMv8-A architecture.

It contains the following sections:

2.1 Assembling SVE code

Use `armclang` with a suitable SVE-enabled target to assemble code containing SVE instructions.

The SVE architectural extension to the ARMv8-A architecture (`armv8-a+sve`) provides SVE instructions. Many of these SVE instructions make use of the new `p` and `z` register classes.

Note

The legacy ARM assembler, `armasm`, does not support SVE instructions.

The following example shows a simple assembly program that includes SVE instructions.

```
// example1.s
    .global main
main:
    mov     x0, 0x90000000
    mov     x8, xzr
    ptrue   p0.s                        //SVE instruction
    fcpy    z0.s, p0/m, #5.00000000     //SVE instruction
    orr     w10, wzr, #0x400
loop:
    st1w    z0.s, p0, [x0, x8, lsl #2]  //SVE instruction
    incw    x8                          //SVE instruction
    whilelt p0.s, x8, x10               //SVE instruction
    b.any   loop                        //SVE instruction
    mov     w0, wzr
    ret
```

To assemble this source file into a binary object file, use `armclang` with an SVE-enabled target:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a+sve example1.s -o
example1.o
```

The command-line options in this example are:

`-c`
: Standard `armclang` option. Instructs the compiler to perform the compilation step, but not the link step.

`--target=aarch64-arm-none-eabi`
: Standard `armclang` option. Instructs the compiler to generate A64 instructions for AArch64 state.

  Note

  SVE is not supported with AArch32 state, so the `--target=aarch64-arm-none-eabi` option is mandatory.

`-march=armv8-a+sve`
: New target argument for the standard `armclang -march` option. Specifies that the compiler targets the ARMv8-A architecture profile with the SVE target feature enabled.

  The default for AArch64 is `-march=armv8-a`, that is the ARMv8-A architecture profile without the SVE extension. You must explicitly specify `+sve` to assemble SVE instructions.

  Valid SVE-enabled targets are:

  - `-march=armv8-a+sve`
  - `-march=armv8.1-a+sve`
  - `-march=armv8.2-a+sve`
  - `-march=armv8.3-a+sve`

`example1.s`
: Standard `armclang` option. Input assembly language file.

`-o example1.o`
: Standard `armclang` option. Output ELF object file.

Related references

2.2 Disassembling SVE object files on page 2-16.
4.1 Compiler options on page 4-40.

Related information

armclang Reference Guide.
armclang Reference Guide: -c option.
armclang Reference Guide: -o option.
armclang Reference Guide: -march option.
armclang Reference Guide: --target option.

2.2 Disassembling SVE object files

Use the `llvm-objdump` tool with the SVE target feature enabled to display the details and contents of an ELF-format binary file. This includes disassembly of the text section of an object containing SVE instructions.

Note

`fromelf` does not support disassembly of object files containing SVE instructions.

To disassemble an ELF-format object file containing SVE instructions, use `llvm-objdump` with the `-disassemble` option and the SVE target feature enabled:

```
llvm-objdump -disassemble -mattr=+sve example1.o

example1.o:     file format ELF64-aarch64-little

Disassembly of section .text:
main:
       0:    00 00 b2 d2    mov        x0, #2415919104
       4:    e8 03 1f aa    mov        x8, xzr
       8:    e0 e3 98 25    ptrue      p0.s
       c:    80 c2 90 05    fmov       z0.s, p0/m, #5.00000000
      10:    ea 03 16 32    orr        w10, wzr, #0x400

loop:
      14:    00 40 48 e5    st1w       {z0.s}, p0, [x0, x8, lsl #2]
      18:    e8 e3 b0 04    incw       x8
      1c:    00 15 aa 25    whilelt    p0.s, x8, x10
      20:    a1 ff ff 54    b.ne       #-12
      24:    e0 03 1f 2a    mov        w0, wzr
      28:    c0 03 5f d6    ret
```

Note

Ensure that the ARM Compiler 6 `bin` directory is earlier in your path than any system installation of `llvm-objdump`.

The command-line options in this example are:

`-disassemble` (or `-d`)
: Disassemble the `.text` section of the object file, displaying assembler mnemonics for the machine instructions.

`-mattr=+sve`
: Enables the SVE target feature. This option is required to correctly disassemble SVE instructions.

`example1.o`
: Input ELF-format object file.

Related concepts

2.1 Assembling SVE code on page 2-14.

2.3 Compiling C and C++ code for SVE-enabled targets

ARM Compiler is an advanced auto-vectorizing compiler for the 64-bit ARMv8-A architecture and supports the SVE Architectural extension.

Generating SVE assembly code from C and C++ code

ARM Compiler can produce annotated assembly, and this is a good first step to see how the auto-vectorizer generates SVE instructions.

The following C program subtracts corresponding elements in two arrays, writing the result to a third array. The three arrays are declared using the `restrict` keyword, indicating to the compiler that they do not overlap in memory.

```
// example1.c
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
  for (int i = 0; i < ARRAYSIZE; i++)
  {
    a[i] = b[i] - c[i];
  }
}

int main()
{
  subtract_arrays(a, b, c);
}
```

Compile the program as follows:

```
armclang -O3 -S --target=aarch64-arm-none-eabi -march=armv8-a+sve -o example1.s
example1.c
```

The output assembly code is saved as `example1.s`. The section of the generated assembly language file containing the compiled `subtract_arrays` function appears as follows:

```
subtract_arrays:                        // @subtract_arrays
// BB#0:
        orr     w9, wzr, #0x400
        mov     x8, xzr
        whilelo p0.s, xzr, x9
.LBB0_1:                                // =>This Inner Loop Header: Depth=1
        ld1w    {z0.s}, p0/z, [x1, x8, lsl #2]
        ld1w    {z1.s}, p0/z, [x2, x8, lsl #2]
        sub     z0.s, z0.s, z1.s
        st1w    {z0.s}, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        b.mi    .LBB0_1
// BB#2:
        ret
```

SVE instructions operate on the `z` and `p` register banks. In this example the inner loop is almost entirely composed of SVE instructions. The auto-vectorizer has converted the scalar loop from the original C source code into a vector loop that is independent of the width of SVE vector registers.

Generating an executable binary from C and C++ code

To generate an executable binary, compile your program without the `-S` option:

```
armclang -O3 -Xlinker "--ro_base=0x80000000" --target=aarch64-arm-none-eabi
-march=armv8-a+sve -o example1 example1.c
```

You can specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary:

```
armclang -O3 -Xlinker "--ro_base=0x80000000" --target=aarch64-arm-none-eabi
-march=armv8-a+sve -o example2 example2a.c example2b.c
```

Note

When compiling binaries to execute on the AEMv8-A Base Fixed Virtual Platform (FVP) base model, use the `-Xlinker "--ro_base=0x80000000"` option to specify the location in memory to load and run the binary. The RAM base address for this FVP is 0x80000000.

These executable binaries are suitable for execution on an SVE-enabled AEMv8-A Base Fixed Virtual Platform (FVP). Binaries are automatically linked against the ARM C/C++ library, which is included as part of the ARM Compiler distribution.

The ARM C/C++ library provides many common C functions. The version used by the ARM Compiler is configured for semihosting. This allows a compiled binary to run on an FVP, and pass I/O operations to the host system, removing the need to run a full operating system within the FVP.

Compiling and linking object files as separate steps

To compile each of your source files individually into an object file, specify the `-c` (compile-only) `armclang` option, and then pass the resulting object files into another invocation of armclang to link them into an executable binary. The `-Xlinker` argument is not required until the final invocation.

```
armclang -O3 --target=aarch64-arm-none-eabi -march=armv8-a+sve -c -o example2a.o example2a.c
armclang -O3 --target=aarch64-arm-none-eabi -march=armv8-a+sve -c -o example2b.o example2b.c
armclang -O3 -Xlinker "--ro_base=0x80000000" --target=aarch64-arm-none-eabi
   -march=armv8-a+sve -o example2 example2a.o example2b.o
```

Related references

4.1 Compiler options on page 4-40.

Related information

Fixed Virtual Platforms, on www.arm.com.
armclang Reference Guide.
armclang Reference Guide: -c option.
armclang Reference Guide: -o option.
armclang Reference Guide: -Xlinker option.
armclang Reference Guide: -O option.
armclang Reference Guide: -march option.
armclang Reference Guide: -S option.
armclang Reference Guide: --target option.

2.4 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP)

Describes how to compile a program with ARM Compiler and then run the resulting binary using the AEMv8-A Base Fixed Virtual Platform (FVP). This demonstrates some basic features and shows how increasing the SVE vector width produces a corresponding performance gain.

Running the FVP

The command to execute a compiled binary through the FVP is fairly complex, but there are only a few elements that can be edited.

The following example shows a complete command-line invocation of the FVP. Most of the lines are required for correct program execution and do not need to be modified. The *italic* elements indicate parameters that can be edited.

```
$FVP_BASE/FVP_Base_AEMv8A-AEMv8A \
  --plugin $FVP_BASE/ScalableVectorExtension.so \
  -C SVE.ScalableVectorExtension.veclen=$VECLEN \
  --quiet \
  --stat \
  -C cluster0.NUM_CORES=1 \
  -C bp.secure_memory=0 \
  -C bp.refcounter.non_arch_start_at_default=1 \
  -C cluster0.cpu0.semihosting-use_stderr=1 \
  -C bp.vis.disable_visualisation=1 \
  -C cluster0.cpu0.semihosting-cmd_line="$CMDLINE" \
  -a cluster0.cpu0=$BINARY
```

Where:

`$FVP_BASE`
: Specifies the path to the FVP.

`$VECLEN`
: Defines the SVE vector width, in units of 64-bit (8 byte) blocks. The maximum value is 32, which corresponds to the architectural maximum SVE vector width of 2048 bits (256 bytes).

  The SVE architecture only supports vector lengths in 128-bit (16 byte increments), so all values of `$VECLEN` should be even. For example, a value of 8 would signify a 512-bit vector width.

`--quiet`
: Specifies that the FVP emits reduced output. For example, if `--quiet` is omitted, `Simulation is started` and `Simulation is terminating` messages are output to signify the start and end of program execution.

`--stat`
: Specifies that the FVP writes a short summary of program execution to standard output following termination (even if `--quiet` is specified).

  This output is of the form:

  ```
  Total instructions executed: 10344
  User time:    0.01 sec
  Kernel time: 0.00 sec
  CPU time:    0.01 sec
  Elapsed clock: 0.00 sec
  ```

`$CMDLINE`
: Specifies the command-line to pass to your program. This is typically of the form `"./binary_name arg1 arg2"`.

`$BINARY`
: Specifies the path to the compiled binary that will be loaded and executed by the FVP.

A sample application

The following sample application contains two vectorizable loops. The first fills the `values` array with floating-point values, and the second calculates the total. The application then performs a `printf` operation, producing output when executed through the FVP.

```
#include <stdio.h>
#define ITERATIONS 8192
float values[ITERATIONS];
void fill()
{
  for (int i = 0; i < ITERATIONS; i++)
  {
    values[i] = (float)i;
  }
}

float reduce() {
  float result = 0.0;
  for (int i = 0; i < ITERATIONS; i++)
  {
    result += values[i];
  }
  return result;
}

int main(int argc, char* argv[]) {
  fill();
  printf("Result was %f\n", reduce());
}
```

To compile this application and create an executable binary:

```
armclang -O3 -Xlinker "--ro_base=0x80000000" --target=aarch64-arm-none-eabi
   -march=armv8-a+sve -o sum sum.c
```

Running the sample application on an FVP

To execute an application using an FVP, it is useful to construct a shell script as follows:

```
#!/bin/bash
# fvp-run.sh
# Usage: fvp-run.sh [veclen] [binary]
#    Executes the specified binary in the FVP, with no command-line
#    arguments.  The SVE register width will be [veclen] x 64 bits. Only
#    even values of veclen are valid.
#
#
# Set the FVP_BASE environment variable to point to the FVP directory.
#
# Set the ARMLMD_LICENSE_FILE environment variable to reference a license
# file or license server with entitlement for the FVP.

VECLEN=$1
CMDLINE=$2

$FVP_BASE/FVP_Base_AEMv8A-AEMv8A \
   --plugin $FVP_BASE/ScalableVectorExtension.so \
   -C SVE.ScalableVectorExtension.veclen=$VECLEN \
   --quiet \
   --stat \
   -C cluster0.NUM_CORES=1 \
   -C bp.secure_memory=0 \
   -C bp.refcounter.non_arch_start_at_default=1 \
   -C cluster0.cpu0.semihosting-use_stderr=1 \
   -C bp.vis.disable_visualisation=1 \
   -C cluster0.cpu0.semihosting-cmd_line="$CMDLINE"  \
   -a cluster0.cpu0=$CMDLINE
```

This script loads and executes a compiled binary with the FVP, configured for a specified vector width.

Running the compiled binary through the FVP generates output of the form:

```
$ ./fvp-run.sh 2 ./sum
terminal_3: Listening for serial connection on port 5000
terminal_2: Listening for serial connection on port 5001
terminal_1: Listening for serial connection on port 5002
terminal_0: Listening for serial connection on port 5003
Result was 33549136.000000
```

```
Total instructions executed:      62090
User time:   0.01 sec
Kernel time: 0.01 sec
CPU time:    0.02 sec
Elapsed clock: 0.00 sec
```

The first line is the command-line invocation, passing a vector width of `2` and the application binary `./sum`. The line starting "Result was" is generated by the application. The remainder of the output is the result of specifying the `--stat` option.

Varying the vector width

Varying the SVE vector width changes the total instruction count. The wider the SVE vector, the fewer instructions are needed to process the array. The following example bash command-line executes the binary with all possible vector widths, extracting and printing the instruction count as returned by the FVP `--stat` option.

```
$ for x in {2..32..2};
  do echo -ne VL=$x\\t;
  ./fvp-run.sh $x ./sum | grep 'instructions' | cut -f2 -d:;
  done
VL=2          62090
VL=4          50826
VL=6          47075
VL=8          45194
VL=10         44072
VL=12         43324
VL=14         42785
VL=16         42378
VL=18         42070
VL=20         41817
VL=22         41619
VL=24         41443
VL=26         41300
VL=28         41179
VL=30         41179
VL=32         41179
```

Related references

4.1 Compiler options on page 4-40.

Related information

armclang Reference Guide.
armclang Reference Guide: -o option.
armclang Reference Guide: -Xlinker option.
armclang Reference Guide: -O option.
armclang Reference Guide: -march option.
armclang Reference Guide: --target option.

Chapter 3
Coding Considerations

Describes best practices for writing code that can encourage the SVE Compiler to produce optimal auto-vectorized output.

It contains the following sections:

3.1 Best practices to enable auto-vectorization

To encourage the SVE Compiler to produce optimal auto-vectorized output, code can be structured and hints provided to inform the compiler of program features that it would otherwise not be able to determine. This allows the compiler to produce optimal auto-vectorized output.

Use the restrict keyword if appropriate

The C99 `restrict` keyword (or the non-standard C/C++ `__restrict__` keyword) indicates to the compiler that a specified pointer does not alias with any other pointers for the lifetime of that pointer. This guidance allows the compiler to vectorize loops more aggressively, since it becomes possible to prove that loop iterations are independent and can be executed in parallel.

If these keywords are used erroneously (that is, if another pointer is used to access the same memory) then the behavior is undefined. It is possible that the results of optimized code will differ from that of its unoptimized equivalent.

Use pragmas

The compiler supports pragmas that you can use to explicitly indicate that loop iterations are completely independent from each other. See *Using pragmas to encourage or suppress auto-vectorization* on page 3-31 for more details and examples.

The loop index variable

Where possible, use `<` conditions rather than `<=` or `!=` when constructing loops. This helps the compiler to prove that a loop terminates before the index variable wraps.

The compiler might also be able to perform more loop optimizations if signed integers are used, because the C standard allows for undefined behavior in the case of signed integer overflow. This is not the case for unsigned integers.

Use the –ffp-mode=fast option if it is safe to do so

As highlighted in the description of the *`-ffp-mode=fast` option* on page 4-40, this can significantly improve the quality of generated code, but it does so at the expense of strict compliance with IEEE and ISO standards for mathematical operations. Ensure that your algorithms are tolerant of potential inaccuracies that could be introduced by the use of this option.

Related references

3.2 Auto-vectorization examples on page 3-24.
4.1 Compiler options on page 4-40.

Related information

armclang Reference Guide.
armclang Reference Guide: -ffast-math option.

3.2 Auto-vectorization examples

Describes example C code for loops that the compiler is capable of vectorizing, with a short description of the interesting features of each loop.

Reductions

Loops that perform a reduction operation, such as the following "running total" calculation, can be vectorized.

If the `-ffp-mode=fast` option is specified, the compiler maintains a vector of per-lane running totals. When the loop completes, it performs a cross-lane reduction operation to efficiently sum all elements into a single scalar value.

Without the `-ffp-mode=fast` option, the compiler can still vectorize, but is likely to produce less efficient code, as it is constrained to perform the summation operation in the same order as the original source code.

```
float reduction(float *restrict a, long count)
{
  float r = 0;
  for (long i = 0; i < count; i++)
  {
    r += a[i];
  }
  return r;
}
```

Strided access

Access to memory does not need to be sequential. In this example every fifth element in two arrays are added together and written to the corresponding element in a destination array.

```
void stride(int *restrict a, int *restrict b, int *restrict c, long count)
{
  for (long i = 0; i < count; i+=5)
  {
    a[i] = b[i] + c[i];
  }
}
```

Scatter and gather

You can use the SVE scatter and gather operations to efficiently auto-vectorize when loop iterations do not have a regular access pattern. In the following example an indices array indicates which elements in the data array should be added together. The compiler loads as many indices elements as can fit in an SVE vector, and then uses them as offsets from a base register, to perform a gather-load operation from the data array.

```
float gather_reduce(float *restrict data, int *restrict indices, long c)
{
  float r = 0;
  for (long i = 0; i < c; i++)
  {
    r += data[indices[i]];
  }
  return r;
}
```

Conditions within the loop body

The predication features of the SVE architectural extension make it possible to efficiently support unpredictable control flow within vectorized loops.

```
float cond_gather_reduce(float *restrict data, int *restrict indices, long count)
{
  float r = 0;
  for (long i = 0; i < count; i++)
  {
    if (indices[i]%2 == 0)
```

```
    {
      r += data[indices[i]];
    }
  }
  return r;
}
```

Related references

3.1 Best practices to enable auto-vectorization on page 3-23.

4.1 Compiler options on page 4-40.

Related information

armclang Reference Guide.

armclang Reference Guide: -ffast-math option.

3.3 Embedding SVE assembly code directly into C and C++ code

Inline assembly (or inline asm) provides a mechanism for inserting hand-written assembly instructions into C and C++ code. This lets you vectorize parts of a function by hand without having to write the entire function in assembly code.

Note

This information assumes that you are familiar with details of the SVE Architecture, including vector-width agnostic registers, predication, and `WHILE` operations.

Using inline assembly rather than writing a separate `.s` file has the following advantages:

- 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 gives the compiler more information about what the assembly code does.
- The compiler can inline the function that contains the assembly code into its 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.

Structure of an inline assembly statement

The compiler supports the GNU form of inline assembly. Note that it does not support the Microsoft form of inline assembly.

More detailed documentation of the `asm` construct is available at *the GCC website*.

Inline assembly statements have the following form:

```
asm ("instructions" : outputs : inputs : side-effects);
```

Where:

instructions
: is a text string that contains AArch64 assembly instructions, with at least one newline sequence \n between consecutive instructions.

outputs
: is a comma-separated list of outputs from the assembly instructions.

inputs
: is a comma-separated list of inputs to the assembly instructions.

side-effects
: is a comma-separated list of effects that the assembly instructions have, besides reading from inputs and writing to outputs.

Additionally, the `asm` keyword might need to be followed by the `volatile` keyword.

Outputs

Each entry in outputs has one of the following forms:

```
[name] "=&register-class" (destination)
[name] "=register-class" (destination)
```

The first form has the register class preceded by =&. This specifies that the assembly instructions might read from one of the inputs (specified in the `asm` statement's inputs section) after writing to the output.

The second form has the register class preceded by =. This specifies that the assembly instructions never read from inputs in this way. Using the second form is an optimization. It allows the compiler to allocate the same register to the output as it allocates to one of the inputs.

Both forms specify that the assembly instructions produce an output that is stored in the C object specified by `destination`. This can be any scalar value that is valid for the left-hand side of a C

assignment. The register-class field specifies the type of register that the assembly instructions require. It can be one of:

`r`
: if the register for this output when used within the assembly instructions is a general-purpose register (x0-x30)

`w`
: if the register for this output when used within the assembly instructions is a SIMD and floating-point register (v0-v31).

It is not possible at present for outputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to the inline assembly block.

It is the responsibility of the compiler to allocate a suitable output register and to copy that register into the `destination` after the `asm` statement is executed. The assembly instructions within the instructions section of the `asm` statement can use one of the following forms to refer to the output value:

`%[name]`
: to refer to an r-class output as `x`*`N`* or a w-class output as `v`*`N`*

`%w[name]`
: to refer to an r-class output as `w`*`N`*

`%s[name]`
: to refer to a w-class output as `s`*`N`*

`%d[name]`
: to refer to a w-class output as `d`*`N`*

In all cases *`N`* represents the number of the register that the compiler has allocated to the output. The use of these forms means that it is not necessary for the programmer to anticipate precisely which register is selected by the compiler. The following example creates a function that returns the value 10. It shows how the programmer is able to use the `%w[res]` form to describe the movement of a constant into the output register without knowing which register is used.

```
int f()
{
  int result;
  asm("movz %w[res], #10" : [res] "=r" (result));
  return result;
}
```

In optimized output the compiler picks the return register (0) for `res`, resulting in the following assembly code:

```
movz w0, #10
ret
```

Inputs

Within an `asm` statement, each entry in the `inputs` section has the form:

```
[name] "operand-type" (value)
```

This construct specifies that the `asm` statement uses the scalar C expression value as an input, referred to within the assembly instructions as `name`. The operand-type field specifies how the input value is handled within the assembly instructions. It can be one of the following:

`r`
: if the input is to be placed in a general-purpose register (x0-x30)

`w`
: if the input is to be placed in a SIMD and floating-point register (v0-v31).

`[`*`output-name`*`]`
: if the input is to be placed in the same register as output *`output-name`*. In this case the `[`*`name`*`]` part of the input specification is redundant and can be omitted. The assembly instructions can use the forms described in the Outputs section above (`%[`*`name`*`]`, `%w[`*`name`*`]`, `%s[`*`name`*`]`, `%d[`*`name`*`]`) to refer to both the input and the output.

`i`
: if the input is an integer constant and is used as an immediate operand. The assembly instructions use `%[name]` in place of immediate operand `#N`, where `N` is the numerical value of `value`.

In the first two cases, it is the responsibility of the compiler to allocate a suitable register and to ensure that it contains `value` on entry to the assembly instructions. The assembly instructions must refer to these registers using the same syntax as for the outputs (`%[name]`, `%w[name]`, `%s[name]`, `%d[name]`).

It is not possible at present for inputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to instructions.

This example shows an `asm` directive with the same effect as the previous example, except that an i-form input is used to specify the constant to be assigned to the result.

```
int f()
{
   int result;
   asm("movz %w[res], %[value]" : [res] "=r" (result) : [value] "i" (10));
   return result;
}
```

Side effects

Many `asm` statements have effects other than reading from inputs and writing to outputs. This is particularly true of `asm` statements that implement vectorized loops, since most such loops read from or write to memory. The `side-effects` section of an `asm` statement tells the compiler what these additional effects are. Each entry must be one of the following:

`"memory"`
: if the `asm` statement reads from or writes to memory. This is necessary even if inputs contain pointers to the affected memory.

`"cc"`
: if the `asm` statement modifies the condition-code flags.

`"xN"`
: if the `asm` statement modifies general-purpose register `N`.

`"vN"`
: if the `asm` statement modifies SIMD and floating-point register `N`.

`"zN"`
: if the `asm` statement modifies SVE vector register `N`. Since SVE vector registers extend the SIMD and floating-point registers, this is equivalent to writing `"vN"`.

`"pN"`
: if the asm statement modifies SVE predicate register `N`.

Use of volatile

Sometimes an `asm` statement might have dependencies and side effects that cannot be captured by the `asm` statement syntax. For example, suppose there are three separate `asm` statements (not three lines within a single `asm` statement), that do the following:

- The first sets the floating-point rounding mode.
- The second executes on the assumption that the rounding mode set by the first statement is in effect.
- The third statement restores the original floating-point rounding mode.

It is important that these statements are executed in order, but the `asm` statement syntax provides no direct method for representing the dependency between them. Instead, each statement must add the keyword `volatile` after `asm`. This prevents the compiler from removing the `asm` statement as dead code, even if the `asm` statement does not modify memory and if its results appear to be unused. The compiler always executes `asm` volatile statements in their original order.

For example:

```
asm volatile ("msr fpcr, %[flags]" :: [flags] "r" (new_fpcr_value));
```

Note

An `asm volatile` statement must still have a valid side effects list. For example, an `asm volatile` statement that modifies memory must still include `"memory"` in the side-effects section.

Labels

The compiler might output a given `asm` statement more than once, either as a result of optimizing the function that contains the `asm` statement or as a result of inlining that function into some of its callers. Therefore, `asm` statements must not define named labels like `.loop`, since if the `asm` statement is written more than once, the output contains more than one definition of label `.loop`. Instead, the assembler provides a concept of relative labels. Each relative label is simply a number and is defined in the same way as a normal label. For example, relative label 1 is defined by:

```
1:
```

The assembly code can contain many definitions of the same relative label. Code that refers to a relative label must add the letter `f` to refer the next definition (`f` is for forward) or the letter `b` (backward) to refer to the previous definition. A typical assembly loop with a pre-loop test would therefore have the following structure. This allows the compiler output to contain many copies of this code without creating any ambiguity.

```
    ...pre-loop test...
    b.none      2f
1:
    ...loop...
    b.any       1b
2:
```

Example

The following example shows a simple function that performs a fused multiply-add operation (x=a·b+c) across four passed-in arrays of a size specified by *n*:

```
void f(double *restrict x, double *restrict a, double *restrict b, double *restrict c,
            unsigned long n)
{
  for (unsigned long i = 0; i < n; ++i)
  {
    x[i] = fma(a[i], b[i], c[i]);
  }
}
```

An `asm` statement that exploited SVE instructions to achieve equivalent behavior might look like the following:

```
void f(double *x, double *a, double *b, double *c, unsigned long n)
{
  unsigned long i;
  asm ("whilelo p0.d, %[i], %[n]               \n\
  1:                                           \n\
        ld1d z0.d, p0/z, [%[a], %[i], lsl #3]  \n\
        ld1d z1.d, p0/z, [%[b], %[i], lsl #3]  \n\
        ld1d z2.d, p0/z, [%[c], %[i], lsl #3]  \n\
        fmla z2.d, p0/m, z0.d, z1.d            \n\
        st1d z2.d, p0, [%[x], %[i], lsl #3]    \n\
        uqincd %[i]                            \n\
        whilelo p0.d, %[i], %[n]               \n\
        b.any 1b"
   : [i] "=&r" (i)
   : "[i]" (0),
     [x] "r" (x),
     [a] "r" (a),
     [b] "r" (b),
     [c] "r" (c),
     [n] "r" (n)
   : "memory", "cc", "p0", "z0", "z1", "z2");
}
```

Note

Keeping the `restrict` qualifiers would be valid but would have no effect.

The input specifier `"[i]" (0)` indicates that the assembly statements take an input 0 in the same register as output `[i]`. In other words, the initial value of `[i]` must be zero. The use of `=&` in the specification of `[i]` indicates that `[i]` cannot be allocated to the same register as `[x]`, `[a]`, `[b]`, `[c]`, or `[n]` (because the assembly instructions use those inputs after writing to `[i]`).

In this example, the C variable `i` is not used after the `asm` statement. In effect the `asm` statement is simply reserving a register that it can use as scratch space. Including `"memory"` in the side effects list indicates that the `asm` statement reads from and writes to memory. The compiler must therefore keep the `asm` statement even though `i` is not used.

3.4 Using pragmas to encourage or suppress auto-vectorization

ARM Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas make use of, and extend, the `pragma clang loop` directives.

For more information about the `pragma clang loop` directives, see *Auto-Vectorization in LLVM, at llvm.org*.

Note

In all the following cases, the pragma only affects the loop statement immediately following it. If your code contains multiple nested loops, you must insert a pragma before each one in order to affect all the loops in the nest.

Encouraging auto-vectorization with pragmas

If the SVE auto-vectorization pass is enabled with `-O2` or above, then by default it examines all loops.

If static analysis of a loop indicates that it might contain dependencies that hinder parallelism, auto-vectorization might not be performed. If you know that these dependencies do not hinder vectorization, you can use the `interleave` directive to indicate this to the compiler by placing the following line immediately before the loop:

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

This pragma indicates to the compiler that the following loop 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.

Note

Use of this pragma does not guarantee auto-vectorization. There might be other reasons why auto-vectorization is not possible or worthwhile for a particular loop.

Ensure that you only use this pragma when it is safe to do so. Using this pragma when there are data dependencies between loop iterations results in incorrect behavior.

For example, consider the following loop, that processes an array `indices`. Each element in `indices` specifies the index into a larger `histogram` array. The referenced element in the `histogram` array is incremented.

```
void update(int *restrict histogram, int *restrict indices, int count)
{
  for (int i = 0; i < count; i++)
  {
    histogram[ indices[i] ]++;
  }
}
```

The compiler is unable to vectorize this loop, because the same index could appear more than once in the `indices` array. Therefore a vectorized version of the algorithm would lose some of the increment operations if two identical indices are processed in the same vector load/increment/store sequence.

However, if the programmer knows that the `indices` array only ever contains unique elements, then it is useful to be able to force the compiler to vectorize this loop. This is accomplished by placing the pragma before the loop:

```
void update(int *restrict histogram, int *restrict indices, int count)
{
  #pragma clang loop vectorize(assume_safety)
  for (int i = 0; i < count; i++)
  {
    histogram[ indices[i] ]++;
  }
}
```

The following table shows the differences between the compiler output for these functions, where the only difference is the presence of the pragma to encourage vectorization:

Table 3-1 Compiler output with and without auto-vectorization

With no pragma	With `#pragma clang loop vectorize(assume_safety)`

`update:`
`  cmp  w2, #1`
`  b.lt  .LBB0_2`
`.LBB0_1:`
`  ldrsw  x8, [x1], #4`
`  sub  w2, w2, #1`
`  lsl  x8, x8, #2`
`  ldr  w9, [x0, x8]`
`  add  w9, w9, #1`
`  str  w9, [x0, x8]`
`  cbnz  w2, .LBB0_1`
`.LBB0_2:`
`  ret`
	`update_unique:`
	`// BB#0:`
	`  subs  w9, w2, #1`
	`  b.lt  .LBB0_3`
	`// BB#1:`
	`  add  x9, x9, #1`
	`  mov  x8, xzr`
	`  whilelo p0.d, xzr, x9`
	
	`.LBB0_2:`
	`  ld1sw  {z0.d}, p0/z, [x1, x8, lsl #2]`
	`  incd  x8`
	`  ld1sw  {z1.d}, p0/z, [x0, z0.d, lsl #2]`
	`  add  z1.d, z1.d, #1`
	`  st1w  {z1.d}, p0, [x0, z0.d, lsl #2]`
	`  whilelo p0.d, x8, x9`
	`  b.mi  .LBB0_2`
	`.LBB0_3:`
	`  ret`

Suppressing auto-vectorization with pragmas

If SVE auto-vectorization is not required for a specific loop, you can disable it or restrict it to only use ARM NEON instructions.

You can suppress auto-vectorization on a specific loop by adding `#pragma clang loop vectorize(disable)` immediately before the loop. In this example, a loop that would be trivially vectorized by the compiler is ignored:

```
void update_unique(int *restrict a, int *restrict b, int count)
{
  #pragma clang loop vectorize(disable)
  for ( int i = 0; i < count; i++ )
  {
    a[i] = b[i] + 1;
  }
}
```

You can also suppress SVE instructions while allowing ARM NEON instructions by adding a `vectorize_style` hint:

`vectorize_style(fixed_width)`
: Prefer fixed-width vectorization, resulting in ARM NEON instructions. For a loop with `vectorize_style(fixed_width)`, the compiler prefers to generate ARM NEON instructions, though SVE instructions may still be used with a fixed-width predicate (such as gather loads or scatter stores).

`vectorize_style(scaled_width)`
: Prefer scaled-width vectorization, resulting in SVE instructions. For a loop with `vectorize_style(scaled_width)`, the compiler prefers SVE instructions but can choose to generate ARM NEON instructions or not vectorize at all.

  This is the default.

For example:

```
void update_unique(int *restrict a, int *restrict b, int count)
{
  #pragma clang loop vectorize(enable) vectorize_style(fixed_width)
  for ( int i = 0; i < count; i++ )
  {
    a[i] = b[i] + 1;
  }
}
```

3.5 Using SVE intrinsics directly in your C code

Intrinsics are C or C++ pseudo-function calls that the compiler replaces with the appropriate SIMD instructions. This lets you use the data types and operations available in the SIMD implementation, while allowing the compiler to handle instruction scheduling and register allocation.

These intrinsics are defined in the *ARM C Language Extensions for SVE* specification.

Introduction

The ARM C language extensions for SVE provide a set of types and accessors for SVE vectors and predicates, and a function interface for all relevant SVE instructions.

The function interface is more general than the underlying architecture, so not every function maps directly to an architectural instruction. The intention is to provide a regular interface and leave the compiler to pick the best mapping to SVE instructions.

The *ARM C Language Extensions for SVE* specification has a detailed description of this interface, and should be used as the primary reference. This section introduces a selection of features to help you get started with the ARM C Language Extensions (ACLE) for SVE.

Note

Please be aware of issue 1758, described in *5.2 Known limitations in SVE support* on page 5-44.

Header file inclusion

Translation units that use the ACLE should first include `arm_sve.h`, guarded by `__ARM_FEATURE_SVE`:

```
#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */
```

All functions and types that are defined in the header file have the prefix `sv`, to reduce the chance of collisions with other extensions.

SVE vector types

`arm_sve.h` defines the following C types to represent values in SVE vector registers. Each type describes the type of the elements within the vector:

svint8_t svuint8_t

svint16_t svuint16_t svfloat16_t

svint32_t svuint32_t svfloat32_t

svint64_t svuint64_t svfloat64_t

For example, svint64_t represents a vector of 64-bit signed integers, and svfloat16_t represents a vector of half-precision floating-point numbers.

SVE predicate type

The extension also defines a single sizeless predicate type `svbool_t`, which has enough bits to control an operation on a vector of bytes.

The main use of predicates is to select elements in a vector. When the elements in the vector have N bytes, only the low bit in each sequence of N predicate bits is significant, as shown in the following table:

Table 3-2 Element selection by predicate type svbool_t

Vector type	Element selected by each svbool_t bit									
svint8_t	0	1	2	3	4	5	6	7	8	...
svint16_t	0		1		2		3		4	...
svint32_t	0				1				2	...
svint64_t	0								1	...

Limitations on how SVE ACLE types can be used

SVE is a vector-length agnostic architecture, allowing an implementation to choose a vector length of any multiple of 128 bits, up to a maximum of 2048 bits. Therefore, the size of SVE ACLE types are unknown at compile time, which limits how these types can be used.

Common situations where SVE types may be used include:

- as the type of an object with automatic storage duration
- as a function parameter or return type
- as the type in a `(type) {value}` compound literal
- as the target of a pointer or reference type
- as a template type argument.

Due to their unknown size at compile time, SVE types may not be used:

- to declare or define a static or thread-local storage variable
- as the type of an array element
- as the operand to a new expression
- as the type of object deleted by a delete expression
- as the argument to `sizeof` and `_Alignof`
- with pointer arithmetic on pointers to SVE objects (this affects the `+`, `-`, `++`, and `--` operators)
- as members of unions, structures and classes
- in standard library containers like `std::vector`.

For a comprehensive list of valid usage, refer to the *ARM C Language Extensions for SVE* specification.

Writing SVE ACLE functions

SVE ACLE functions have the form:

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

where the function is built using the following:

`base`
: For most functions, this is the lowercase name of the SVE instruction. Sometimes, letters indicating the type or size of data being operated on are dropped, where it can be implied from the argument types.

  Unsigned extending loads add a `u` to indicate that the data will be zero extended, to more explicitly differentiate them from their signed equivalent.

`disambiguator`
: This field distinguishes between different forms of a function, for example:
  - To distinguish between addressing modes
  - To distinguish forms that take a scalar rather than a vector as the final argument.

`type0 type1 ...`
: A list of types for vectors and predicates, starting with the return type then with each argument type. For example, `_s8`, `_u32`, and `_f32`, which represent signed 8-bit integer, an unsigned 32-bit integer and single precision 32-bit float types, respectively.

  Predicate types are represented by `_b8`, `_b16` and so on, for predicates suitable for 8-bit and 16-bit types respectively. A predicate type suitable for all element types is represented by `_b`. Where a type is not needed to disambiguate between variants of a base function, it is omitted.

`predication`
: This suffix describes the inactive elements in the result of a predicated operation. It can be one of the following:
  - z – Zero predication: Set all inactive elements of the result to zero.
  - m – Merge predication: copy all inactive elements from the first vector argument.
  - x – 'Don't care' predication. Use this form when you do not care about the inactive elements. The compiler is then free to choose between zeroing, merging, or unpredicated forms to give the best code quality, but gives no guarantee of what data, if any, will be left in inactive elements.

Addressing modes

Load, store, prefetch, and ADR functions have arguments that describe the memory area being addressed. The first addressing argument is the base – either a single pointer to an element type, or a 32-bit or 64-bit vector of addresses. The second argument, when present, offsets the base (or bases) by some number of bytes, elements, or vectors. This offset argument can be an immediate constant value, a scalar argument, or a vector of offsets.

Not every combination of the above options exists. The following table gives examples of some common addressing mode disambiguators, and describes how to interpret the address arguments:

Table 3-3 Common addressing mode disambiguators

Disambiguator	Interpretation
_u32base	The base argument is a vector of unsigned 32-bit addresses.
_u64base	The base argument is a vector of unsigned 64-bit addresses.
_s32offset _s64offset _u32offset _u64offset	The offset argument is a vector of byte offsets. These offsets are signed or unsigned 32-bit or 64-bit numbers.
_s32index _s64index _u32index _u64index	The offset argument is a vector of element-sized indices. These indices are signed or unsigned 32-bit or 64-bit numbers.
_offset	The offset argument is a scalar, and should be treated as a byte offset.
_index	The offset argument is a scalar, and should be treated as an index into an array of elements.
_vnum	The offset argument is a scalar, and should be treated an index into an array of SVE vectors.

In the following example, the address of element `i` is `&base[indices[i]]`.

```
svuint32_t svld1_gather_[s32]index[_u32]
    (svbool_t pg, const uint32_t *base, svint32_t indices)
```

Operations involving vectors and scalars

All arithmetic functions that take two vector inputs have an alternative form that takes a vector and a scalar. Conceptually, this scalar is duplicated across a vector, and that vector is used as the second vector argument.

Similarly, arithmetic functions that take three vector inputs have an alternative form that takes two vectors and one scalar.

To differentiate these forms, the disambiguator `_n` is added to the form that takes a scalar.

Short forms

Sometimes, it is possible to omit part of the full name, and still uniquely identify the correct form of a function, by inspecting the argument types. Where this is possible, these simplified forms are provided as aliases to their fully named equivalents, and will be used for preference in the rest of this document.

In the *ARM C Language Extensions for SVE* specification, the portion that can be removed is enclosed in square brackets. For example `svclz[_s16]_m` has the full name `svclz_s16_m`, and an overloaded alias, `svclz_m`.

Example – Naïve step-1 daxpy

daxpy is a BLAS (Basic Linear Algebra Subroutines) subroutine that operates on two arrays of double precision floating-point numbers. A slice is taken of each of these arrays. For each element in these slices, an element (x) in the first array is multiplied by a constant (a), then added to the element (y) from the second array. The result is stored back to the second array at the same index.

This example presents a step-1 *daxpy* implementation, where the indices of x and y start at 0 and increment by 1 each iteration. A C code implementation might look like this:

```
void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    for (int64_t i = 0; i < n; ++i) {
        dy[i] = dx[i] * da + dy[i];
    }
}
```

Here is an ACLE equivalent:

```
void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);                          // [1]
    do
        {
            svfloat64_t dx_vec = svld1(pg, &dx[i]);                     // [2]
            svfloat64_t dy_vec = svld1(pg, &dy[i]);                     // [2]
            svst1(pg, &dy[i], svmla_x(pg, dy_vec, dx_vec, da));         // [3]
            i += svcntd();                                              // [4]
            pg = svwhilelt_b64(i, n);                                   // [1]
        }
    while (svptest_any(svptrue_b64(), pg));                                   // [5]
}
```

The following steps explain this example:

[1] - Initialize a predicate register to control the loop. `_b64` specifies a predicate for 64-bit elements. Conceptually, this operation creates an integer vector starting at `i` and incrementing by 1 in each subsequent lane. The predicate lane is active if this value is less than `n`. Therefore, this loop is safe, if inefficient, even if $n \leq 0$. The same operation is used at the bottom of the loop, to update the predicate for the next iteration.

[2] - Load some values into an SVE vector, guarded by the loop predicate. Lanes where this predicate is false do not perform any load (and so will not generate a fault), and set the result value to 0.0. The number of lanes that are loaded depends on the vector width, which is only known at runtime.

[3] - Perform a floating-point multiply-add operation, and pass the result to a store. The `_x` on the MLA indicates we don't care about the result for inactive lanes. This gives the compiler maximum flexibility in choosing the most efficient instruction. The result of this operation is stored at address `&dy[i]`, guarded by the loop predicate. Lanes where the predicate is false are not stored, and so the value in memory will retain its prior value.

[4] - Increment `i` by the number of double-precision lanes in the vector.

[5] - `ptest` returns true if any lane of the (newly updated) predicate is active, which causes control to return to the start of the while loop if there is any work left to do.

"Ideal" assembler output:

```
daxpy_1_1:
    MOV Z2.D, D0            // da
    MOV X3, #0              // i
    WHILELT P0.D, X3, X0    // i, n
loop:
    LD1D Z1.D, P0/Z, [X1, X3, LSL #3]
    LD1D Z0.D, P0/Z, [X2, X3, LSL #3]
    FMLA Z0.D, P0/M, Z1.D, Z2.D
    ST1D Z0.D, P0, [X2, X3, LSL #3]
    INCD X3                 // i
    WHILELT P0.D, X3, X0    // i, n
    B.ANY loop
    RET
```

Example – Naïve general daxpy

This example presents a general *daxpy* implementation, where the indices of x and y start at 0 and are then incremented by unknown (but loop-invariant) strides each iteration.

```
void daxpy(int64_t n, double da, double *dx, int64_t incx,
           double *dy, int64_t incy)
{
  svint64_t incx_vec = svindex_s64(0, incx);                                  // [1]
  svint64_t incy_vec = svindex_s64(0, incy);                                  // [1]
  int64_t i = 0;
  svbool_t pg = svwhilelt_b64(i, n);                                          // [2]
  do
    {
      svfloat64_t dx_vec = svld1_gather_index(pg, dx, incx_vec);              // [3]
      svfloat64_t dy_vec = svld1_gather_index(pg, dy, incy_vec);              // [3]
      svst1_scatter_index(pg, dy, incy_vec, svmla_x(pg, dy_vec, dx_vec, da)); // [4]
      dx += incx * svcntd();                                                  // [5]
      dy += incy * svcntd();                                                  // [5]
      i += svcntd();                                                          // [6]
      pg = svwhilelt_b64(i, n);                                               // [2]
    }
  while (svptest_any(svptrue_b64(), pg));                                     // [7]
}
```

The following steps explain this example:

[1] - For each of `x` and `y`, initialize a vector of indices, starting at 0 for the first lane and incrementing by `incx` and `incy` respectively in each subsequent lane.

[2] - Initialize or update the loop predicate.

[3] - Load a vector's worth of values, guarded by the loop predicate. Lanes where this predicate is false do not perform any load (and so will not generate a fault), and set the result value to 0.0. This time, a base + vector-of-indices gather load, is used to load the required non-consecutive values.

[4] - Perform a floating-point multiply-add operation, and pass the result to a store. This time, the base + vector-of-indices scatter store is used to store each result in the correct index of the `dy[]` array.

[5] - Instead of using `i` to calculate the load address, increment the base pointer, by multiplying the vector length by the stride.

[6] - Increment `i` by the number of double-precision lanes in the vector.

[7] - Test the loop predicate to work out whether there is any more work to do, and loop back if appropriate.

Chapter 4
Reference

Provides reference information about ARM Compiler.

It contains the following section:

4.1 Compiler options

Describes the subset of ARM Compiler command-line options most likely to be of interest to users developing code to take advantage of SVE.

See the *armclang Reference Guide* for information about the full range of supported options.

Options controlling compiler operation

`-S`

Outputs assembly code, rather than object code.

Produces a text `.s` file containing annotated assembly code.

`-c`

Performs the compilation step, but does not invoke armlink to perform the link step.

Produces an ELF object `.o` file. To later link object files into an executable binary, run `armclang` again, passing in the object files.

`-o` *file*

Specifies the name of the output file.

`-std=`*lang*

Compiles for the specified language standard, for example `-std=c90` or `-std=c++98`.

See the *armclang Reference Guide* for details of the supported variants, and the differences between them.

`--target=`*arch-vendor-os-abi*

Generates code for the selected target.

ARM Compiler supports both AArch32 and AArch64 targets. However, SVE is an extension to ARMv8-A AArch64. Therefore the only supported target for this release is `aarch64-arm-none-eabi`

`-Xlinker opt`

Specifies a linker command-line option to pass to the linker when a link step is being performed after compilation.

When compiling binaries to execute on the AEMv8-A Base Fixed Virtual Platform (FVP) base model, use this option to specify the location in memory to load and run the binary. The RAM base address for this FVP is 0x80000000. You must therefore specify the `-Xlinker "--ro_base=0x80000000"` option for any `armclang` invocation that performs the link stage.

Options controlling compiler optimization

`-march=`*name*`[+[no]`*feature*`]`

Targets an architecture profile, generating generic code that runs on any processor of that architecture.

Append the `+sve` feature to enable SVE, or omit to disable.

For example, `-march=armv8.2+sve` enables the ARMv8.2 architecture profile plus SVE support.

Use `-march=list` to display a list of all the supported architectures for your target.

`-O`*level*
: Specifies the level of optimization to use when compiling source files.

  SVE auto-vectorization occurs only at the `-O2` and `-O3` levels. Auto-vectorization is identical at both levels, however `-O3` results in higher general code optimization.

  The `-Ofast` option is equivalent to `-O3 -ffp-mode=fast`, and can produce faster code if fast math optimizations are appropriate for your application.

  The `-Omax` option is equivalent to `-Ofast` plus other aggressive optimizations. It specifically targets performance optimization.

`-ffp-mode=fast`
: Enable aggressive floating-point optimizations.

  The compiler can perform code optimizations and transformations that, although arithmetically correct, might not be in strict compliance with IEEE or ISO rules concerning mathematical operations.

  This option can produce significantly faster code. For example, it allows the compiler to use the SVE `FADDV` instruction to perform fast parallel additions across a vector. The `FADDV` instruction is faster than the `FADDA` instruction because `FADDA` performs all additions across the vector in strict sequence. Take care to ensure that your algorithms do not depend on strict IEEE or ISO mathematical rules. The optimizations applied here are not guaranteed to produce bitwise identical results when compared with less aggressively optimized code.

Informational options

`--help`
: Describes the most common options supported by ARM Compiler.

`--vsn, --version`
: Displays version information and license details.

Related information

armclang Reference Guide.

Chapter 5
Troubleshooting

Provides general troubleshooting advice relating to SVE functionality in this release of ARM Compiler.

It contains the following sections:

5.1 General troubleshooting advice

Provides general advice for troubleshooting problems with ARM Compiler.

SVE instructions are not generated

If your compiled code does not contain SVE instructions, check the following:

- Ensure that ARM Compiler was used to compile the code.

  If your output files are human-readable assembly code (that is, you specified the `-S` option), examine the `.s` output files and look for lines containing the `.ident` keyword. If the output is produced by ARM Compiler, lines similar to the following are displayed:

  ```
  .ident    "Component: ARM Compiler 6.6.0: armclang [5c50d600]"
  ```

  If your output files are ELF object files, use `fromelf` to inspect the `.comment` section of an object file. If the output is produced by ARM Compiler, lines similar to the following are displayed:

  ```
  $ fromelf -v main.o | grep armclang
    Component: ARM Compiler 6.6.0: armclang [5c50d600]
  ```

- Check you specified an optimization level of `-O2` or greater.

  The SVE auto-vectorizer is only invoked if the optimization level specified is `-O2` or greater. Ensure that your compiler invocation includes a flag such as `-O2` to enable SVE auto-vectorization.

  See *Compiler options* on page 4-40 for details of the compiler flags that control the behavior of the SVE auto-vectorizer.
- Consider modifying source code.

  When you have confirmed that the SVE Compiler is being invoked, and there is a loop that is not being vectorized, see *Best practices to enable auto-vectorization* on page 3-23 for examples and hints on how to encourage the compiler to auto-vectorize your code.

Fatal error: L6450U: Cannot find library m

Code that makes use of the mathematical functions declared in `math.h` usually requires a `-lm` flag when compiling on a POSIX environment such as Linux, to link in the Linux implementation of those functions for that host system. For this reason, you might find that such a flag is added to your scripts or makefiles.

ARM Compiler is designed to run in a bare metal environment, and automatically includes implementations of these functions, and so no such flag is necessary.

If you encounter error L6450U, remove the `-lm` flag from your makefiles or build scripts.

Related references

4.1 Compiler options on page 4-40.

3.1 Best practices to enable auto-vectorization on page 3-23.

Related information

armclang Reference Guide: -O option.

armclang Reference Guide: -S option.

5.2 Known limitations in SVE support

The following are known limitations of ARM Compiler when compiling for SVE architectures:

- Link-Time Optimization (the `-flto` option) is not supported when compiling for SVE architectures.
- Maximum optimization (the `-Omax` option) is not supported when compiling for SVE architectures.