Learn the architecture - Compiling for Neon with auto-vectorization

Version 1.0

Non-Confidential

Issue 00

102525_0100_00_en

Learn the architecture - Compiling for Neon with auto-vectorization

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	18 June 2019	Non-Confidential	First release

Proprietary Notice

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.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

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

- Neon-enabled open source libraries such as the Arm Compute Library provide one of the easiest ways to take advantage of Neon.
- Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Neon.
- Neon intrinsics are function calls that the compiler replaces with appropriate Neon instructions. This gives you direct, low-level access to the exact Neon instructions you want, all from C/C++ code.
- For very high performance, hand-coded Neon assembler can be an alternative approach for experienced programmers.

This guide shows how to use the auto-vectorization features in Arm Compiler 6 to automatically generate code that contains Armv8 Advanced SIMD instructions. It contains a number of examples to explore Neon code generation and highlights coding best practices that help the compiler produce the best results.

This guide will be useful to everyone developing for Arm, and will be especially useful for those who want to use Neon technology without having to program in assembly.

At the end of this guide you will have achieved the following:

- You will know which Arm Compiler command line options enable Advanced SIMD code generation.
- You will be able to write C/C++ code which exploits various optimization features of Arm Compiler 6.
- You will know where to find the documentation for different compilers.

If you are not already familiar with Neon, you should read Introducing Neon for Armv8-A before starting this guide.

The examples in this guide use Arm Compiler 6, designed for embedded application development running on bare-metal devices. If you do not already have access to Arm Compiler 6, it is included in the 30-day free trial of Arm Development Studio Gold Edition.

Even though this guide uses Arm Compiler 6, you can easily adapt the examples for other compilers. You will need to consult your compiler documentation to find out the equivalent compiler options to use in the examples. Auto-vectorizing compilers that can generate Neon code include:

- Arm Compiler 6, designed for embedded application development running on bare-metal devices. This is the compiler used in this guide's examples.
- Arm C/C++ Compiler, designed for Linux user space application development, originally for High Performance Computing.
- LLVM-clang, the open source LLVM-based toolchain.

- GCC, the open source GNU toolchain.

2. Why rely on the compiler for auto-vectorization?

Writing hand-optimized assembly kernels or C code containing Neon intrinsics provides a high level of control over the Neon code in your software. However, these methods can result in significant portability and engineering complexity costs.

In many cases a high quality compiler can generate code which is just as good, but requires significantly less design time. The process of allowing the compiler to automatically identify opportunities in your code to use Advanced SIMD instructions is called auto-vectorization.

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

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

Auto-vectorizing compilers include Arm Compiler 6, Arm C/C++ Compiler, LLVM-clang, and GCC.

The benefits of relying on compiler auto-vectorization include the following:

- 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, whereas optimizing an assembly program requires deep knowledge of the target hardware.

Auto-vectorization might not be the right choice in all situations, however:

- While source code can be architecture agnostic, it may have to be compiler specific to get the best code-generation.
- Small changes in a high-level language or the compiler options can result in significant and unpredictable changes in generated code.

Using the compiler to generate Neon code will be appropriate for most projects. Other methods for exploiting Neon only become necessary when the generated code does not deliver the necessary performance, or when particular hardware features are not supported by high-level languages. For example, configuring system registers to control floating-point functionality must be performed in assembly code.

3. Compiling for Neon with Arm Compiler 6

To enable automatic vectorization you must specify appropriate compiler options to do the following:

- Target a processor that has a Neon capabilities.
- Specify an optimization level that includes auto-vectorization.

In addition, specifying the `-Rpass=loop` compiler option displays useful diagnostic information from the compiler about how it optimized particular loops. This information includes vectorization width and interleave count.

Note

`-Rpass=loop` is a [COMMUNITY] feature of Arm Compiler.

Specifying a Neon-capable target

Neon is required in all standard Armv8-A implementations, so targeting any Armv8-A architecture or processor will allow the generation of Neon code.

If you only want to run code on one particular processor, you can target that specific processor. Performance is optimized for the micro-architectural specifics of that processor. However code is only guaranteed to run on that processor.

If you want your code to run on a wide range of processors, you can target an architecture. Generated code runs on any processor implementation of that target architecture, but performance might be impacted.

To target Armv8-A AArch64 state:

```
armclang --target=aarch64-arm-none-eabi
```

To target the Cortex-A53 in AArch32 state:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53
```

For the older Armv7 architecture, where Neon was optional, you can use the `-mcpu`, `-march` and `-mfpu` options to specify that Neon is available.

Specifying an auto-vectorizing optimization level

Arm Compiler 6 provides a wide range of optimization levels, selected with the `-O` option:

Option	Meaning	Auto-vectorization
-O0	Minimum optimization	Never

Option	Meaning	Auto-vectorization
-O1	Restricted optimization	Disabled by default.
-O2	High optimization	Enabled by default.
-O3	Very high optimization	Enabled by default.
-Os	Reduce code size, balancing code size against code speed.	Enabled by default.
-Oz	Smallest possible code size	Enabled by default.
-Ofast	Optimize for high performance beyond -O3	Enabled by default.
-Omax	Optimize for high performance beyond -Ofast	Enabled by default.

See Selecting optimization options, in the Arm Compiler User Guide and -O, in the Arm Compiler armclang Reference Guide for more details about these options.

Auto-vectorization is enabled by default at optimization level `-O2` and higher. The `-fno-vectorize` option lets you disable auto-vectorization.

At optimization level `-O1`, auto-vectorization is disabled by default. The `-fvectorize` option lets you enable auto-vectorization.

At optimization level `-O0`, auto-vectorization is always disabled. If you specify the `-fvectorize` option, the compiler ignores it.

4. Example: vector addition

Let's look at how we can use compiler options to auto-vectorize and optimize a simple C program.

1. Create a new file `vec_add.c` containing the following function. This function adds two arrays of 32-bit floating-point values.

```
void vec_add(float *vec_A, float *vec_B, float *vec_C, int len_vec) {
          int i;
          for (i=0; i<len_vec; i++) {
                  vec_C[i] = vec_A[i] + vec_B[i];
          }
  }
```

2. Compile the code, without using auto-vectorization:

```
armclang --target=aarch64-arm-none-eabi -g -c -O1 vec_add.c
```

3. Disassemble the resulting object file to see the generated instructions:

```
fromelf --disassemble vec_add.o -o disassembly_vec_off.txt
```

The disassembled code looks similar to this:

```
vec_add                  ; Alternate entry point
          CMP      w3,#1
          B.LT     |L3.36|
          MOV      w8,w3
  |L3.12|
          LDR      s0,[x0],#4
          LDR      s1,[x1],#4
          SUBS     x8,x8,#1
          FADD     s0,s0,s1
          STR      s0,[x2],#4
          B.NE     |L3.12|
  |L3.36|
          RET
```

Here we can see the label name `vec_add` for the function, followed by the generated assembly instructions that make up the function. The `FADD` instruction performs the core part of the operation, but the code is not making use of Neon as only one addition operation is performed at a time. We can see this because the `FADD` instruction is operating on the scalar registers `s0` and `s1`.

4. Re-compile the code, this time using auto-vectorization:

```
armclang --target=aarch64-arm-none-eabi -g -c -O1 vec_add.c -fvectorize
```

5. Disassemble the resulting object file to see the generated instructions:

```
fromelf --disassemble vec_add.o -o disassembly_vec_on.txt
```

The disassembled code looks similar to this:

```
vec_add                   ; Alternate entry point
          CMP      w3,#1
          B.LT     |L3.184|
          CMP      w3,#4
          MOV      w8,w3
          MOV      x9,xzr
          B.CC     |L3.140|
          LSL      x10,x8,#2
          ADD      x12,x0,x10
          ADD      x11,x2,x10
          CMP      x12,x2
          ADD      x10,x1,x10
          CSET     w12,HI
          CMP      x11,x0
          CSET     w13,HI
          CMP      x10,x2
          CSET     w10,HI
          CMP      x11,x1
          AND      w12,w12,w13
          CSET     w11,HI
          TBNZ     w12,#0,|L3.140|
          AND      w10,w10,w11
          TBNZ     w10,#0,|L3.140|
          AND      x9,x8,#0xfffffffc
          MOV      x10,x9
          MOV      x11,x2
          MOV      x12,x1
          MOV      x13,x0
  |L3.108|
          LDR      q0,[x13],#0x10
          LDR      q1,[x12],#0x10
          SUBS     x10,x10,#4
          FADD     v0.4S,v0.4S,v1.4S
          STR      q0,[x11],#0x10
          B.NE     |L3.108|
          CMP      x9,x8
          B.EQ     |L3.184|
  |L3.140|
          LSL      x12,x9,#2
          ADD      x10,x2,x12
          ADD      x11,x1,x12
          ADD      x12,x0,x12
          SUB      x8,x8,x9
  |L3.160|
          LDR      s0,[x12],#4
          LDR      s1,[x11],#4
          SUBS     x8,x8,#1
          FADD     s0,s0,s1
          STR      s0,[x10],#4
          B.NE     |L3.160|
  |L3.184|
          RET
```

SLP auto-vectorization has been successful, as we can see from the instruction `FADD v0.4S,v0.4S,v1.4S` which performs an addition on four 32-bit floats packed into a SIMD register. However this has come at significant cost to code size as it must detect cases where the SIMD width is not a divisor of the array length. Such increases in code size may or may not be acceptable depending on the project and target hardware. This may be tolerable for a phone application where the change in code size is insignificant compared with the available memory, but could be unacceptable for an embedded application with a small amount of RAM.

A complete code listing is included below. Compile and disassemble at different optimization levels to see the effect on the generated code.

5. Example:function in a loop

Sometimes changes to source code are unavoidable if you want to use particular optimization features of the compiler. This can occur when the code is too complex for the compiler to auto-vectorize, or when you want to override the compiler's decisions about how to optimize a particular piece of code.

1. Create a new file `cubed.c` containing the following function. This function calculates the cubes of an array of values.

```
double cubed(double x) {
          return x*x*x;
  }

  void vec_cubed(double *x_vec, double *y_vec, int len_vec) {
          int i;
          for (i=0; i<len_vec; i++) {
                  y_vec[i] = cubed(x_vec[i]);
          }
  }
```

2. Compile the code, using auto-vectorization:

```
armclang --target=aarch64-arm-none-eabi -g -c -O1 -fvectorize cubed.c
```

3. Disassemble the resulting object file to see the generated instructions:

```
fromelf --disassemble cubed.o -o disassembly.txt
```

The disassembled code looks similar to this:

```
cubed                  ; Alternate entry point
          FMUL     d1,d0,d0
          FMUL     d0,d1,d0
          RET

          AREA ||.text.vec_cubed||, CODE, READONLY, ALIGN=2

  vec_cubed                  ; Alternate entry point
          STP      x21,x20,[sp,#-0x20]!
          STP      x19,x30,[sp,#0x10]
          CMP      w2,#1
          B.LT     |L4.48|
          MOV      x19,x1
          MOV      x20,x0
          MOV      w21,w2
  |L4.28|
          LDR      d0,[x20],#8
          BL       cubed
          SUBS     x21,x21,#1
          STR      d0,[x19],#8
          B.NE     |L4.28|
  |L4.48|
          LDP      x19,x30,[sp,#0x10]
          LDP      x21,x20,[sp],#0x20
          RET
```

There are a number of issues in this code:

- The compiler has not performed loop or SLP vectorization, or inlined our cubed function.
- The code needs to perform checks on the input pointers to verify that the arrays do not overlap.

These issues can be fixed in a number of ways, such as compiling at a higher optimization level, but let's focus on what code changes can be made without altering the compiler options.

4. Add the following macros and qualifiers to the code to can override some of the compiler's decisions.

- `__attribute__((always_inline))` is an Arm Compiler extension which indicates that the compiler always attempts to inline the function. In this example, not only is the function inlined, but the compiler can also perform SLP vectorization.

Before inlining, the cubed function works with scalar doubles only, so there is no need or way of performing SLP vectorization on this function by itself.

When the cubed function is inlined, the compiler can detect that its operations are performed on arrays and vectorize the code with the available ASIMD instructions.

- `restrict` is a standard C/C++ keyword that indicates to the compiler that a given array corresponds to a unique region of memory. This eliminates the need for run-time checks for overlapping arrays.
- `#pragma clang loop interleave_count(X)` is a Clang language extension that lets you control auto-vectorization by specifying a vector width and interleaving count. This pragma is a [COMMUNITY] feature of Arm Compiler.

A complete reference to the vectorization macros can be found in the clang documentation.

```
__always_inline double cubed(double x) {
        return x*x*x;
  }

  void vec_cubed(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(2)
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed(x_vec[i]);
        }
  }
```

5. Compile and disassemble with the same commands we used earlier. This produces the following code:

```
vec_cubed                        ; Alternate entry point
          CMP      w2,#1
          B.LT     |L4.132|
          CMP      w2,#4
          MOV      w8,w2
          B.CS     |L4.28|
          MOV      x9,xzr
          B        |L4.92|
  |L4.28|
          AND      x9,x8,#0xfffffffc
```

```
        ADD      x10,x0,#0x10
        ADD      x11,x1,#0x10
        MOV      x12,x9
|L4.44|
        LDP      q0,q1,[x10,#-0x10]
        ADD      x10,x10,#0x20
        SUBS     x12,x12,#4
        FMUL     v2.2D,v0.2D,v0.2D
        FMUL     v3.2D,v1.2D,v1.2D
        FMUL     v0.2D,v0.2D,v2.2D
        FMUL     v1.2D,v1.2D,v3.2D
        STP      q0,q1,[x11,#-0x10]
        ADD      x11,x11,#0x20
        B.NE     |L4.44|
        CMP      x9,x8
        B.EQ     |L4.132|
|L4.92|
        LSL      x11,x9,#3
        ADD      x10,x1,x11
        ADD      x11,x0,x11
        SUB      x8,x8,x9
|L4.108|
        LDR      d0,[x11],#8
        SUBS     x8,x8,#1
        FMUL     d1,d0,d0
        FMUL     d0,d0,d1
        STR      d0,[x10],#8
        B.NE     |L4.108|
|L4.132|
        RET
```

This disassembly shows that the inlining, SLP vectorization, and loop vectorization have been successful. Using the restrict pointers has eliminated run-time overlap checks.

The code size has increased slightly, due to the loop tail which handles any remaining iterations when the total loop count is not a multiple of four (the effective unroll depth). The loop unroll depth is two and is the SLP width is two, so the effective unroll depth is four. In the next step we'll look at an optimization we can make if we know the loop count will always be a multiple of four.

6. Let us assume our loop count will always be a multiple of four. We can communicate this to the compiler by masking off the lower bits of the loop counter:

```
void vec_cubed(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(1)
        for (i=0; i<(len_vec & ~3); i++) {
                y_vec[i] = cubed_i(x_vec[i]);
        }
}
```

7. Compile and disassemble with the same commands we used earlier. This produces the following code:

```
vec_cubed                  ; Alternate entry point
        AND      w8,w2,#0xfffffffc
        CMP      w8,#1
        B.LT     |L13.40|
        MOV      w8,w8
|L13.16|
        LDR      q0,[x0],#0x10
        SUBS     x8,x8,#2
```

```
        FMUL     v1.2D,v0.2D,v0.2D
        FMUL     v0.2D,v0.2D,v1.2D
        STR      q0,[x1],#0x10
        B.NE     |L13.16|
|L13.40|
        RET
```

The code size is reduced, because the compiler knows it no longer has to test for and deal with any remaining iterations that were not a multiple of four. Promising to the compiler that the data we supply will always be a multiple of the vector length has produced optimized code.

This example is simple enough that compiling at `-O2` will perform all of these optimizations with no code changes, but more complex pieces of code might require this type of tuning to get the most from the compiler.

A full code listing is included below. You can compile and disassemble at a variety of optimization levels and unroll depths to observe the compiler's auto-vectorization behavior.

Full source code example: function in a loop

```
/*
 * Copyright (C) Arm Limited, 2019 All rights reserved.
 *
 * The example code is provided to you as an aid to learning when working
 * with Arm-based technology, including but not limited to programming tutorials.
 * Arm hereby grants to you, subject to the terms and conditions of this Licence,
 * a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
 * to use and copy the Software solely for the purpose of demonstration and
 * evaluation.
 *
 * You accept that the Software has not been tested by Arm therefore the Software
 * is provided "as is", without warranty of any kind, express or implied. In no
 * event shall the authors or copyright holders be liable for any claim, damages
 * or other liability, whether in action or contract, tort or otherwise, arising
 * from, out of or in connection with the Software or the use of Software.
 */

#include <stdio.h>

void vec_init(double *vec, int len_vec, double init_val) {
        int i;
        for (i=0; i<len_vec; i++) {
                vec[i] = init_val*i - len_vec/2;
        }
}

void vec_print(double *vec, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                printf("%f, ", vec[i]);
        }
        printf("\n");
}

double cubed(double x) {
        return x*x*x;
}

void vec_cubed(double *x_vec, double *y_vec, int len_vec) {
        int i;
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed(x_vec[i]);
        }
}
```

```
__attribute__((always_inline)) double cubed_i(double x) {
        return x*x*x;
}

void vec_cubed_opt(double *restrict x_vec, double *restrict y_vec, int len_vec) {
        int i;
        #pragma clang loop interleave_count(1)
        for (i=0; i<len_vec; i++) {
                y_vec[i] = cubed_i(x_vec[i]);
        }
}


int main() {
        int N = 10;
        double X[N];
        double Y[N];

        vec_init(X, N, 1);
        vec_print(X, N);
        vec_cubed(X, Y, 10);
        vec_print(Y, N);
        vec_cubed_opt(X, Y, 10);
        vec_print(Y, N);
        return 0;
}
```

6. Coding best practices for auto-vectorization

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

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

Arm recommends modifying your source code implementation to eliminate these situations.

For example, a necessary condition for auto-vectorization is that the number of iterations in the loop size must be known at the start of the loop. Break conditions mean the loop size may not be knowable at the start of the loop, which will prevent auto-vectorization. If it is not possible to completely avoid a break condition, it may be worthwhile breaking up the loops into multiple vectorizable and non-vectorizable parts.

A full discussion of the compiler directives used to control vectorization of loops for can be found in the LLVM-Clang documentation, but the two most important are:

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

These pragmas are hints to the compiler to perform SLP and Loop vectorization respectively. They are [COMMUNITY] features of Arm Compiler.

More detailed guides covering auto-vectorization are available for the Arm C/C++ Compiler Linux user space compiler, although many of the points will apply across LLVM-Clang variants:

- Arm C/C++ Compiler: Coding best practice for auto-vectorization
- Arm C/C++ Compiler: Using pragmas to control auto-vectorization

7. Check your knowledge

The following questions help you test your knowledge:

What is Neon?

Neon is the implementation of the Advanced SIMD extension to the Arm architecture. All processors compliant with the Armv8-A architecture (for example, the Cortex-A76 or Cortex-A57) include Neon. In the programmer's view, Neon provides an additional 32 128-bit registers with instructions that operate on 8, 16, 32, or 64 bit lanes within these registers.

How do you enable Neon code generation with Arm Compiler?

Target AArch64 with `--target=aarch64-arm-none-eabi` and specify a suitable optimization level, such as `-O1 -fvectorize` or `-O2` and higher.

Suppose the Arm compiler automatically unrolls a loop to a depth of two. How would you force the compiler to unroll to a depth of four?

`#pragma clang loop interleave_count(4)` will achieve this, applying only to that particular loop.

How can you best write source code to assist the compiler optimizations?

Consider the following function when compiled with the `-O1` compiler option:

```
float vec_dot(float *vec_A, float *vec_B, int len_vec) {
          float ret = 0;
          int i;
          for (i=0; i<len_vec; i++) {
                  ret += vec_A[i]*vec_B[i];
          }
          return ret;
  }
```

You could make the following changes to assist the compiler optimizations:

- Compile at `-O2` or higher, or with `-fvectorize`.
- Specify `#pragma clang loop vectorize(enable)` before the loop as a hint to the compiler.
- Note that we are not modifying the vectors during the procedure so adding the `restrict` keyword will do nothing here; it doesn't matter if the input arrays overlap.
- SLP vectorization comes with an increased code in this case. This may be acceptable depending on hardware limits and expected input array length.

Here is the optimized source code:

```
float vec_dot(float *vec_A, float *vec_B, int len_vec) {
          float ret = 0;
          int i;
          #pragma clang loop vectorize(enable)
          for (i=0; i<len_vec; i++) {
                  ret += vec_A[i]*vec_B[i];
          }
          return ret;
  }
```

8. Related information

Here are some resources related to material in this guide:

- Arm Compiler 6 documentation provides information about the bare-metal compiler.
- Arm C/C++ Compiler documentation provides information about the Linux user space compiler.
- The LLVM-clang documentation provides information about the open source LLVM-based toolchain.
- The GCC documentation provides information about the open source GNU toolchain.
- The Architecture Exploration Tools let you investigate and learn more about the Advanced SIMD instruction set.
- The Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile provides a complete specification of the Advanced SIMD instruction set.
- The Optimizing C Code with Neon Intrinsics guide shows you how to use Neon intrinsics in your C, or C++, code to take advantage of the Advanced SIMD technology in the Armv8 architecture.