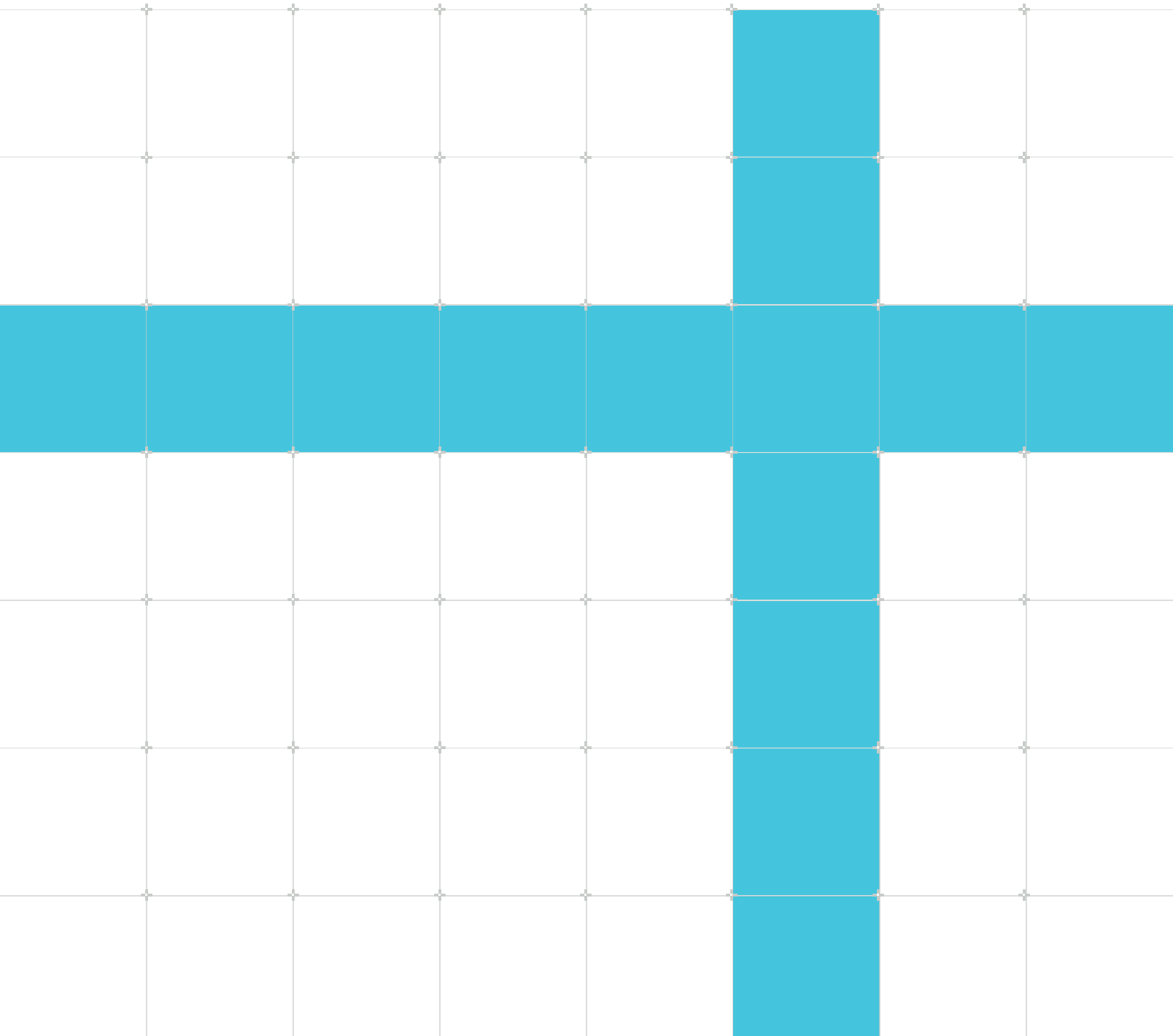
Learn the architecture - Optimizing C code with Neon intrinsics

2.1

Issue 01
102467_0201_01_en



Learn the architecture - Optimizing C code with Neon intrinsics

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	11 March 2022	Non-Confidential	Initial release
0200-01	23 March 2023	Non-Confidential	Added new chapter about collision detection
0201-01	21 September 2023	Non-Confidential	Added two new images about matrix multiplication

Proprietary Notice

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

This 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. The simple examples show how to use these intrinsics and provide an opportunity to explain their purpose.

Intended audience

Low-level software engineers, library writers, and other developers wanting to use Advanced SIMD technology will find this guide useful.

At the end of this guide there is a Check Your Knowledge section to test whether you have understood the following key concepts:

- To know what Neon is, and understand the different ways of using Neon
- To know the basics of using Neon intrinsics in the C language
- To know where to find the Neon intrinsics reference, and the Neon instruction set

2. What is Neon?

Neon is the implementation of Arm's Advanced SIMD architecture.

The purpose of Neon is to accelerate data manipulation by providing:

- Thirty-two 128-bit vector registers, each capable of containing multiple lanes of data.
- SIMD instructions to operate simultaneously on those multiple lanes of data.

Applications that can benefit from Neon technology include multimedia and signal processing, 3D graphics, speech, image processing, or other applications where fixed and floating-point performance is critical.

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, or C ++ code.
- For very high performance, hand-coded Neon assembler can be the best approach for experienced programmers.

In this guide we focus on using the Neon intrinsics for AArch64, but they can also be compiled for AArch32. For more information about AArch32 Neon see Introducing Neon for Armv8-A.

3. Why use Neon intrinsics?

Intrinsics are functions whose precise implementation is known to a compiler. The Neon intrinsics are a set of C and C++ functions defined in `arm_neon.h` which are supported by the Arm compilers and GCC. These functions let you use Neon without having to write assembly code directly, since the functions themselves contain short assembly kernels which are inlined into the calling code. Additionally, register allocation and pipeline optimization are handled by the compiler so many difficulties faced by the assembly programmer are avoided.

See the Neon Intrinsics Reference for a list of all the Neon intrinsics. The Neon intrinsics engineering specification is contained in the Arm C Language Extensions (ACLE).

Using the Neon intrinsics has a number of benefits:

- Powerful: Intrinsics give the programmer direct access to the Neon instruction set without the need for hand-written assembly code.
- Portable: Hand-written Neon assembly instructions might need to be rewritten for different target processors. C and C++ code containing Neon intrinsics can be compiled for a new target or a new execution state (for example, migrating from AArch32 to AArch64) with minimal or no code changes.
- Flexible: The programmer can exploit Neon when needed or use C/C++ when it isn't needed, while avoiding many low-level engineering concerns.

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

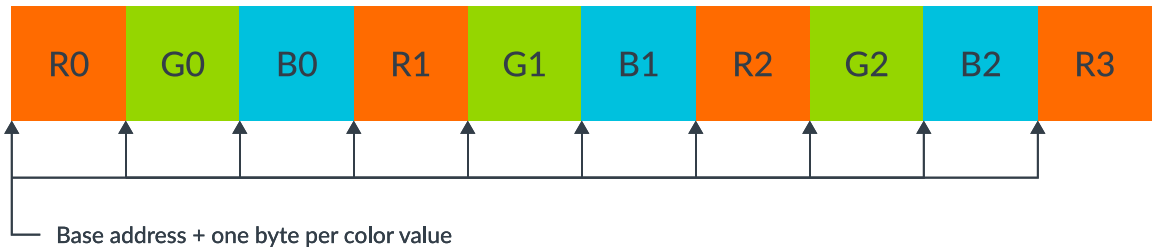
- There is a steeper learning curve to use Neon intrinsics than importing a library or relying on a compiler.
- Hand-optimized assembly code might offer the greatest scope for performance improvement even if it is more difficult to write.

We look at examples where we reimplement some C functions using Neon intrinsics. The examples chosen do not reflect the full complexity of their application, but they illustrate the use of intrinsics and act as a starting point for more complex code.

4. Example - RGB deinterleaving

Consider a 24-bit RGB image where the image is an array of pixels, each with a red, blue, and green element. In memory, this could appear as shown in the following diagram:

Figure 4-1: RGB image pixel array



Because the RGB data is interleaved, accessing and manipulating the three separate color channels presents a problem to the programmer. In simple circumstances we could write our own single color channel operations by applying the modulo 3 to the interleaved RGB values. However, for more complex operations, such as Fourier transforms, it would make more sense to extract and split the channels.

We have an array of RGB values in memory and we want to deinterleave them and place the values in separate color arrays. A C procedure to do this might look like this:

```
void rgb_deinterleave_c(uint8_t *r, uint8_t *g, uint8_t *b, uint8_t *rgb, int
 len_color) {
    /*
     * Take the elements of "rgb" and store the individual colors "r", "g", and "b".
     */
    for (int i=0; i < len_color; i++) {
        r[i] = rgb[3*i];
        g[i] = rgb[3*i+1];
        b[i] = rgb[3*i+2];
    }
}
```

But there is an issue. Compiling with Arm Compiler 6 at optimization level -O3 (very high optimization) and examining the disassembly shows no Neon instructions or registers are being used. Each individual 8-bit value is stored in a separate 64-bit general registers. Considering the full width Neon registers are 128 bits wide, which could each hold 16 of our 8-bit values in the example, rewriting the solution to use Neon intrinsics should give us good results.

```
void rgb_deinterleave_neon(uint8_t *r, uint8_t *g, uint8_t *b, uint8_t *rgb, int
 len_color) {
    /*
     * Take the elements of "rgb" and store the individual colors "r", "g", and "b"
     */
    int num8x16 = len_color / 16;
    uint8x16x3_t intlv_rgb;
```

```
    for (int i=0; i < num8x16; i++) {
        intlv_rgb = vld3q_u8(rgb+3*16*i);
        vst1q_u8(r+16*i, intlv_rgb.val[0]);
        vst1q_u8(g+16*i, intlv_rgb.val[1]);
        vst1q_u8(b+16*i, intlv_rgb.val[2]);
    }
}
```

In this example we have used the following types and intrinsics:

Code element	What is it?	Why are we using it?
`uint8x16_t`	An array of 16 8-bit unsigned integers.	One `uint8x16_t` fits into a 128-bit register. We can ensure there are no wasted register bits even in C code.
`uint8x16x3_t`	A struct with three `uint8x16_t` elements.	A temporary holding area for the current color values in the loop.
`vld3q_u8(...)`	A function which returns a `uint8x16x3_t` by loading a contiguous region of 3*16 bytes of memory. Each byte loaded is placed one of the three `uint8x16_t` arrays in an alternating pattern.	At the lowest level, this intrinsic guarantees the generation of an LD3 instruction, which loads the values from a given address into three Neon registers in an alternating pattern.
`vst1q_u8(...)`	A function which stores a `uint8x16_t` at a given address.	It stores a full 128-bit register full of byte values.

The full source code above can be compiled and disassembled on an Arm machine using the following commands:

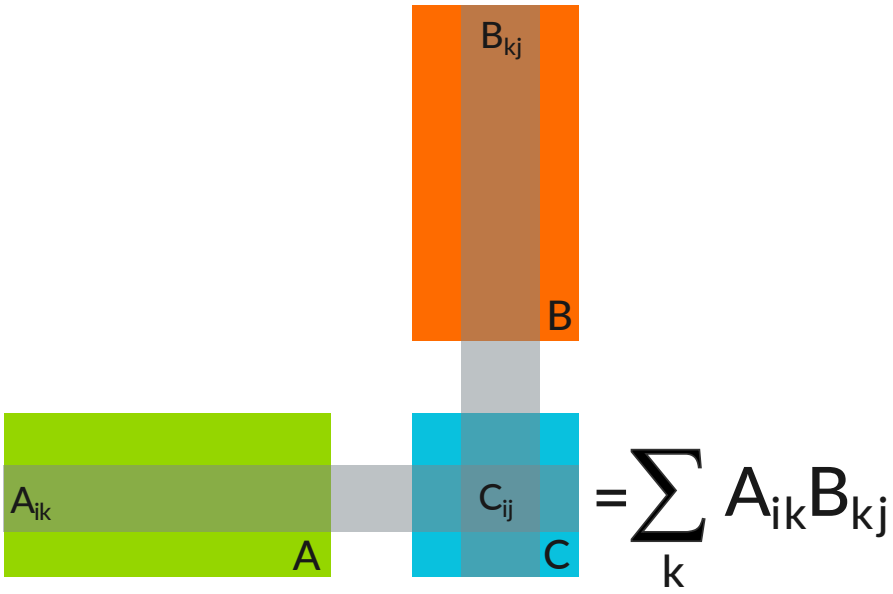
```
gcc -g -o3 rgb.c -o exe_rgb_o3
objdump -d exe_rgb_o3 > disasm_rgb_o3
```

If you don't have access to Arm-based hardware, you can use Arm DS-5 Community Edition and the Armv8-A Foundation Platform.

5. Example - matrix multiplication

Matrix multiplication is an operation performed in many data intensive applications. It is made up of groups of arithmetic operations which are repeated in a straightforward way, as the following diagram shows:

Figure 5-1: Matrix multiplication



The matrix multiplication process is as follows:

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

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

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

We have assumed a column-major layout of the matrices in memory. That is, an `n x m` matrix `M` is represented as an array `M_array` where `Mij = M_array[n*j + i]`.

This code is suboptimal, since it does not make full use of Neon. We can begin to improve it by using intrinsics, but let's tackle a simpler problem first by looking at small, fixed-size matrices before moving on to larger matrices.

The following code uses intrinsics to multiply two `4x4` matrices. Since we have a small and fixed number of values to process, all of which can fit into the processor's Neon registers at once, we can completely unroll the loops.

```
void matrix_multiply_4x4_neon(float32_t *A, float32_t *B, float32_t *C) {
        // these are the columns A
        float32x4_t A0;
        float32x4_t A1;
        float32x4_t A2;
        float32x4_t A3;

        // these are the columns B
        float32x4_t B0;
        float32x4_t B1;
        float32x4_t B2;
        float32x4_t B3;

        // these are the columns C
        float32x4_t C0;
        float32x4_t C1;
        float32x4_t C2;
        float32x4_t C3;

        A0 = vld1q_f32(A);
        A1 = vld1q_f32(A+4);
        A2 = vld1q_f32(A+8);
        A3 = vld1q_f32(A+12);

        // Zero accumulators for C values
        C0 = vmovq_n_f32(0);
        C1 = vmovq_n_f32(0);
        C2 = vmovq_n_f32(0);
        C3 = vmovq_n_f32(0);

        // Multiply accumulate in 4x1 blocks, i.e. each column in C
        B0 = vld1q_f32(B);
        C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
        C0 = vfmaq_laneq_f32(C0, A1, B0, 1);
        C0 = vfmaq_laneq_f32(C0, A2, B0, 2);
        C0 = vfmaq_laneq_f32(C0, A3, B0, 3);
        vst1q_f32(C, C0);

        B1 = vld1q_f32(B+4);
        C1 = vfmaq_laneq_f32(C1, A0, B1, 0);
        C1 = vfmaq_laneq_f32(C1, A1, B1, 1);
        C1 = vfmaq_laneq_f32(C1, A2, B1, 2);
        C1 = vfmaq_laneq_f32(C1, A3, B1, 3);
        vst1q_f32(C+4, C1);

        B2 = vld1q_f32(B+8);
        C2 = vfmaq_laneq_f32(C2, A0, B2, 0);
        C2 = vfmaq_laneq_f32(C2, A1, B2, 1);
        C2 = vfmaq_laneq_f32(C2, A2, B2, 2);
        C2 = vfmaq_laneq_f32(C2, A3, B2, 3);
        vst1q_f32(C+8, C2);

        B3 = vld1q_f32(B+12);
        C3 = vfmaq_laneq_f32(C3, A0, B3, 0);
```

```
        C3 = vfmaq_laneq_f32(C3, A1, B3, 1);
        C3 = vfmaq_laneq_f32(C3, A2, B3, 2);
        C3 = vfmaq_laneq_f32(C3, A3, B3, 3);
        vst1q_f32(C+12, C3);
}
```

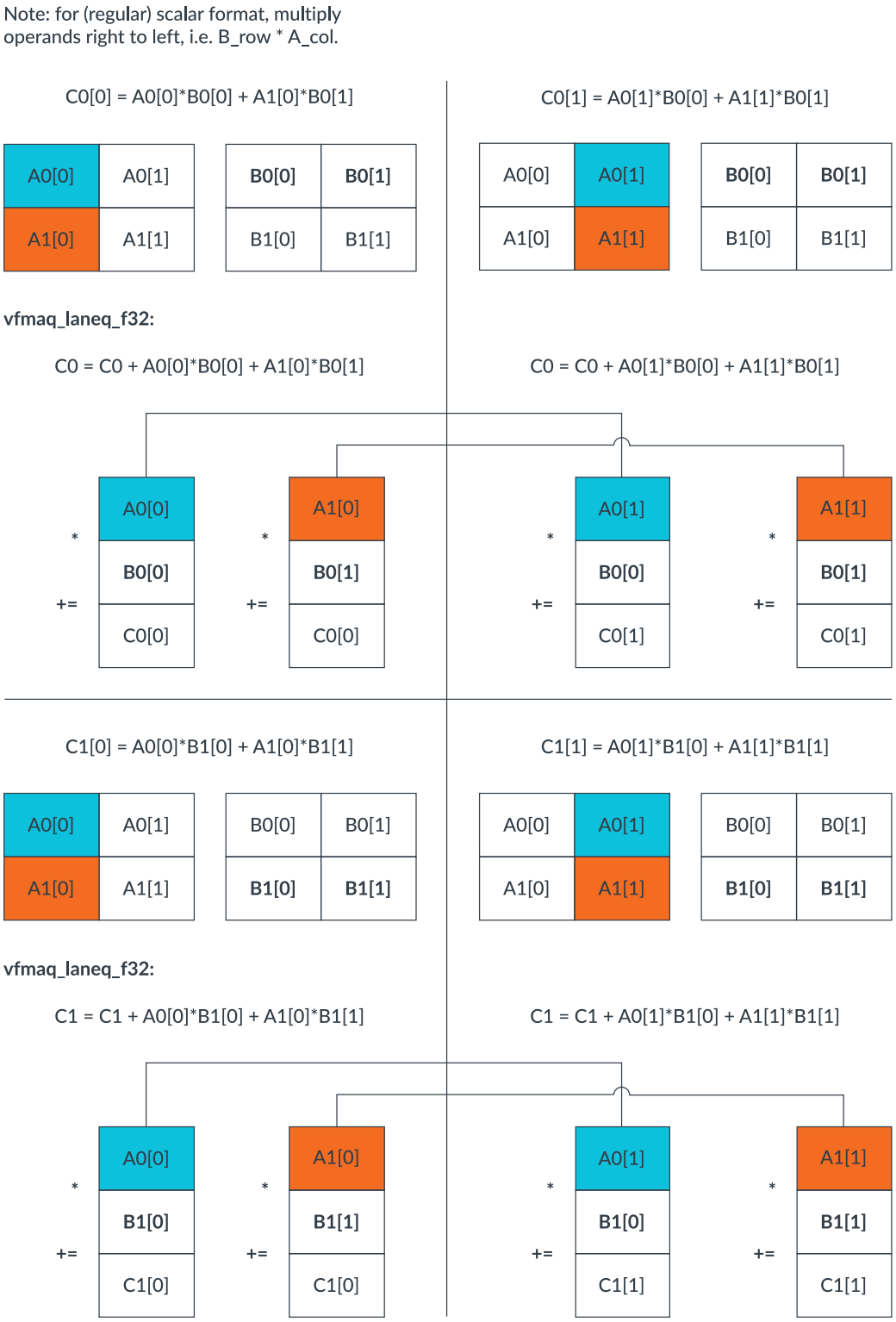
We have chosen to multiply fixed size `4x4` matrices for a few reasons:

- Some applications need `4x4` matrices specifically, for example graphics or relativistic physics.
- The Neon vector registers hold four 32-bit values, so matching the program to the architecture makes it easier to optimize.
- We can take this `4x4` kernel and use it in a more general one.

Let's summarize the intrinsics that have been used here:

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

Figure 5-2: Matrix A multiplied by matrix B, using Neon



Now that we can multiply a `4x4` matrix, we can multiply larger matrices by treating them as blocks of 4x4 matrices. A flaw with this approach is that it only works with matrix sizes which are a multiple of four in both dimensions, but by padding any matrix with zeroes you can use this method without changing it.

The code for a more general matrix multiplication is listed below. The structure of the kernel has changed very little, with the addition of loops and address calculations being the major changes. As in the `4x4` kernel we have used unique variable names for the columns of B, even though we could have used one variable and reloaded. This acts as a hint to the compiler to assign different registers to these variables, which enables the processor to complete the arithmetic instructions for one column while waiting on the loads for another.

```
void matrix_multiply_neon(float32_t  *A, float32_t  *B, float32_t *C, uint32_t n,
 uint32_t m, uint32_t k) {
        /*
         * Multiply matrices A and B, store the result in C.
         * It is the user's responsibility to make sure the matrices are compatible.
         */

        int A_idx;
        int B_idx;
        int C_idx;

        // these are the columns of a 4x4 sub matrix of A
        float32x4_t A0;
        float32x4_t A1;
        float32x4_t A2;
        float32x4_t A3;

        // these are the columns of a 4x4 sub matrix of B
        float32x4_t B0;
        float32x4_t B1;
        float32x4_t B2;
        float32x4_t B3;

        // these are the columns of a 4x4 sub matrix of C
        float32x4_t C0;
        float32x4_t C1;
        float32x4_t C2;
        float32x4_t C3;

        for (int i_idx=0; i_idx<n; i_idx+=4 {
            for (int j_idx=0; j_idx<m; j_idx+=4){
                 // zero accumulators before matrix op
                 c0=vmovq_n_f32(0);
                 c1=vmovq_n_f32(0);
                 c2=vmovq_n_f32(0);
                 c3=vmovq_n_f32(0);
                 for (int k_idx=0; k_idx<k; k_idx+=4){
                      // compute base index to 4x4 block
                      a_idx = i_idx + n*k_idx;
                      b_idx = k*j_idx k_idx;

                      // load most current a values in row
                      A0=vld1q_f32(A+A_idx);
                      A1=vld1q_f32(A+A_idx+n);
                      A2=vld1q_f32(A+A_idx+2*n);
                      A3=vld1q_f32(A+A_idx+3*n);

                      // multiply accumulate 4x1 blocks, i.e. each column C
                      B0=vld1q_f32(B+B_idx);
                      C0=vfmaq_laneq_f32(C0,A0,B0,0);
                      C0=vfmaq_laneq_f32(C0,A1,B0,1);
                      C0=vfmaq_laneq_f32(C0,A2,B0,2);
                      C0=vfmaq_laneq_f32(C0,A3,B0,3);
```

```
                    B1=vld1q_f32(B+B_idx+k);
                    C1=vfmaq_laneq_f32(C1,A0,B1,0);
                    C1=vfmaq_laneq_f32(C1,A1,B1,1);
                    C1=vfmaq_laneq_f32(C1,A2,B1,2);
                    C1=vfmaq_laneq_f32(C1,A3,B1,3);

                    B2=vld1q_f32(B+B_idx+2*k);
                    C2=vfmaq_laneq_f32(C2,A0,B2,0);
                    C2=vfmaq_laneq_f32(C2,A1,B2,1);
                    C2=vfmaq_laneq_f32(C2,A2,B2,2);
                    C2=vfmaq_laneq_f32(C2,A3,B3,3);

                    B3=vld1q_f32(B+B_idx+3*k);
                    C3=vfmaq_laneq_f32(C3,A0,B3,0);
                    C3=vfmaq_laneq_f32(C3,A1,B3,1);
                    C3=vfmaq_laneq_f32(C3,A2,B3,2);
                    C3=vfmaq_laneq_f32(C3,A3,B3,3);
                }
   //Compute base index for stores
   C_idx = n*j_idx + i_idx;
   vstlq_f32(C+C_idx, C0);
   vstlq_f32(C+C_idx+n,C1);
   vstlq_f32(C+C_idx+2*n,C2);
   vstlq_f32(C+C_idx+3*n,C3);
  }
 }
}
```

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

- Fewer arithmetic instructions for a given matrix multiplication, since we are leveraging the Advanced SIMD technology with full register packing. Pure C code generally does not do this.
- `FMLA` instead of `FMUL` instructions. As specified by the intrinsics.
- Fewer loop iterations. When used properly intrinsics allow loops to be unrolled easily.
- However, there are unnecessary loads and stores due to memory allocation and initialization of data types (for example, `float32x4_t`) which are not used in the pure C code.

Full source code example

The full source code for this example is as follows:

```
/*
 * 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>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include <math.h>

#include <arm_neon.h>

#define BLOCK_SIZE 4


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

void matrix_multiply_neon(float32_t  *A, float32_t  *B, float32_t *C, uint32_t n,
 uint32_t m, uint32_t k) {
        /*
         * Multiply matrices A and B, store the result in C.
         * It is the user's responsibility to make sure the matrices are compatible.
         */

        int A_idx;
        int B_idx;
        int C_idx;

        // these are the columns of a 4x4 sub matrix of A
        float32x4_t A0;
        float32x4_t A1;
        float32x4_t A2;
        float32x4_t A3;

        // these are the columns of a 4x4 sub matrix of B
        float32x4_t B0;
        float32x4_t B1;
        float32x4_t B2;
        float32x4_t B3;

        // these are the columns of a 4x4 sub matrix of C
        float32x4_t C0;
        float32x4_t C1;
        float32x4_t C2;
        float32x4_t C3;

        for (int i_idx=0; i_idx<n; i_idx+=4) {
                for (int j_idx=0; j_idx<m; j_idx+=4) {
                        // Zero accumulators before matrix op
                        C0 = vmovq_n_f32(0);
                        C1 = vmovq_n_f32(0);
                        C2 = vmovq_n_f32(0);
                        C3 = vmovq_n_f32(0);
                        for (int k_idx=0; k_idx<k; k_idx+=4) {
                                // Compute base index to 4x4 block
                                A_idx = i_idx + n*k_idx;
                                B_idx = k*j_idx + k_idx;

                                // Load most current A values in row
                                A0 = vld1q_f32(A+A_idx);
                                A1 = vld1q_f32(A+A_idx+n);
                                A2 = vld1q_f32(A+A_idx+2*n);
                                A3 = vld1q_f32(A+A_idx+3*n);

                                // Multiply accumulate in 4x1 blocks, i.e. each
 column in C
                                B0 = vld1q_f32(B+B_idx);
                                C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
```

```
                                C0 = vfmaq_laneq_f32(C0, A1, B0, 1);
                                C0 = vfmaq_laneq_f32(C0, A2, B0, 2);
                                C0 = vfmaq_laneq_f32(C0, A3, B0, 3);

                                B1 = vld1q_f32(B+B_idx+k);
                                C1 = vfmaq_laneq_f32(C1, A0, B1, 0);
                                C1 = vfmaq_laneq_f32(C1, A1, B1, 1);
                                C1 = vfmaq_laneq_f32(C1, A2, B1, 2);
                                C1 = vfmaq_laneq_f32(C1, A3, B1, 3);

                                B2 = vld1q_f32(B+B_idx+2*k);
                                C2 = vfmaq_laneq_f32(C2, A0, B2, 0);
                                C2 = vfmaq_laneq_f32(C2, A1, B2, 1);
                                C2 = vfmaq_laneq_f32(C2, A2, B2, 2);
                                C2 = vfmaq_laneq_f32(C2, A3, B2, 3);

                                B3 = vld1q_f32(B+B_idx+3*k);
                                C3 = vfmaq_laneq_f32(C3, A0, B3, 0);
                                C3 = vfmaq_laneq_f32(C3, A1, B3, 1);
                                C3 = vfmaq_laneq_f32(C3, A2, B3, 2);
                                C3 = vfmaq_laneq_f32(C3, A3, B3, 3);
                        }
                        // Compute base index for stores
                        C_idx = n*j_idx + i_idx;
                        vst1q_f32(C+C_idx, C0);
                        vst1q_f32(C+C_idx+n, C1);
                        vst1q_f32(C+C_idx+2*n, C2);
                        vst1q_f32(C+C_idx+3*n, C3);
                }
        }
}

void matrix_multiply_4x4_neon(float32_t *A, float32_t *B, float32_t *C) {
        // these are the columns A
        float32x4_t A0;
        float32x4_t A1;
        float32x4_t A2;
        float32x4_t A3;

        // these are the columns B
        float32x4_t B0;
        float32x4_t B1;
        float32x4_t B2;
        float32x4_t B3;

        // these are the columns C
        float32x4_t C0;
        float32x4_t C1;
        float32x4_t C2;
        float32x4_t C3;

        A0 = vld1q_f32(A);
        A1 = vld1q_f32(A+4);
        A2 = vld1q_f32(A+8);
        A3 = vld1q_f32(A+12);

        // Zero accumulators for C values
        C0 = vmovq_n_f32(0);
        C1 = vmovq_n_f32(0);
        C2 = vmovq_n_f32(0);
        C3 = vmovq_n_f32(0);

        // Multiply accumulate in 4x1 blocks, i.e. each column in C
        B0 = vld1q_f32(B);
        C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
        C0 = vfmaq_laneq_f32(C0, A1, B0, 1);
        C0 = vfmaq_laneq_f32(C0, A2, B0, 2);
        C0 = vfmaq_laneq_f32(C0, A3, B0, 3);
        vst1q_f32(C, C0);

        B1 = vld1q_f32(B+4);
```

```
        C1 = vfmaq_laneq_f32(C1, A0, B1, 0);
        C1 = vfmaq_laneq_f32(C1, A1, B1, 1);
        C1 = vfmaq_laneq_f32(C1, A2, B1, 2);
        C1 = vfmaq_laneq_f32(C1, A3, B1, 3);
        vst1q_f32(C+4, C1);

        B2 = vld1q_f32(B+8);
        C2 = vfmaq_laneq_f32(C2, A0, B2, 0);
        C2 = vfmaq_laneq_f32(C2, A1, B2, 1);
        C2 = vfmaq_laneq_f32(C2, A2, B2, 2);
        C2 = vfmaq_laneq_f32(C2, A3, B2, 3);
        vst1q_f32(C+8, C2);

        B3 = vld1q_f32(B+12);
        C3 = vfmaq_laneq_f32(C3, A0, B3, 0);
        C3 = vfmaq_laneq_f32(C3, A1, B3, 1);
        C3 = vfmaq_laneq_f32(C3, A2, B3, 2);
        C3 = vfmaq_laneq_f32(C3, A3, B3, 3);
        vst1q_f32(C+12, C3);
}

void print_matrix(float32_t *M, uint32_t cols, uint32_t rows) {
        for (int i=0; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                        printf("%f ", M[j*rows + i]);
                }
                printf("\n");
        }
        printf("\n");
}

void matrix_init_rand(float32_t *M, uint32_t numvals) {
        for (int i=0; i<numvals; i++) {
                M[i] = (float)rand()/(float)(RAND_MAX);
        }
}

void matrix_init(float32_t *M, uint32_t cols, uint32_t rows, float32_t val) {
        for (int i=0; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                        M[j*rows + i] = val;
                }
        }
}

bool f32comp_noteq(float32_t a, float32_t b) {
        if (fabs(a-b) < 0.000001) {
                return false;
        }
        return true;
}

bool matrix_comp(float32_t *A, float32_t *B, uint32_t rows, uint32_t cols) {
        float32_t a;
        float32_t b;
        for (int i=0; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                        a = A[rows*j + i];
                        b = B[rows*j + i];

                        if (f32comp_noteq(a, b)) {
                                printf("i=%d, j=%d, A=%f, B=%f\n", i, j, a, b);
                                return false;
                        }
                }
        }
        return true;
}

int main() {
        uint32_t n = 2*BLOCK_SIZE; // rows in A
```

```
        uint32_t m = 2*BLOCK_SIZE; // cols in B
        uint32_t k = 2*BLOCK_SIZE; // cols in a and rows in b

        float32_t A[n*k];
        float32_t B[k*m];
        float32_t C[n*m];
        float32_t D[n*m];
        float32_t E[n*m];

        bool c_eq_asm;
        bool c_eq_neon;

        matrix_init_rand(A, n*k);
        matrix_init_rand(B, k*m);
        matrix_init(C, n, m, 0);

        print_matrix(A, k, n);
        print_matrix(B, m, k);
        //print_matrix(C, n, m);

        matrix_multiply_c(A, B, E, n, m, k);
        printf("C\n");
        print_matrix(E, n, m);
        printf("==============================\n");

        matrix_multiply_neon(A, B, D, n, m, k);
        printf("Neon\n");
        print_matrix(D, n, m);
        c_eq_neon = matrix_comp(E, D, n, m);
        printf("Neon equal to C? %d\n", c_eq_neon);
        printf("==============================\n");
}
```

The full source code above can be compiled and disassembled on an Arm machine using the following commands:

```
gcc -g -o3 matrix.c -o exe_matrix_o3
objdump -d exe_ matrix _o3 > disasm_matrix_o3
```

If you don't have access to Arm-based hardware, you can use Arm DS-5 Community Edition and the Armv8-A Foundation Platform.

6. Example - collision detection

This example shows how you can use Neon intrinsics to vectorize a simple collision detection algorithm.

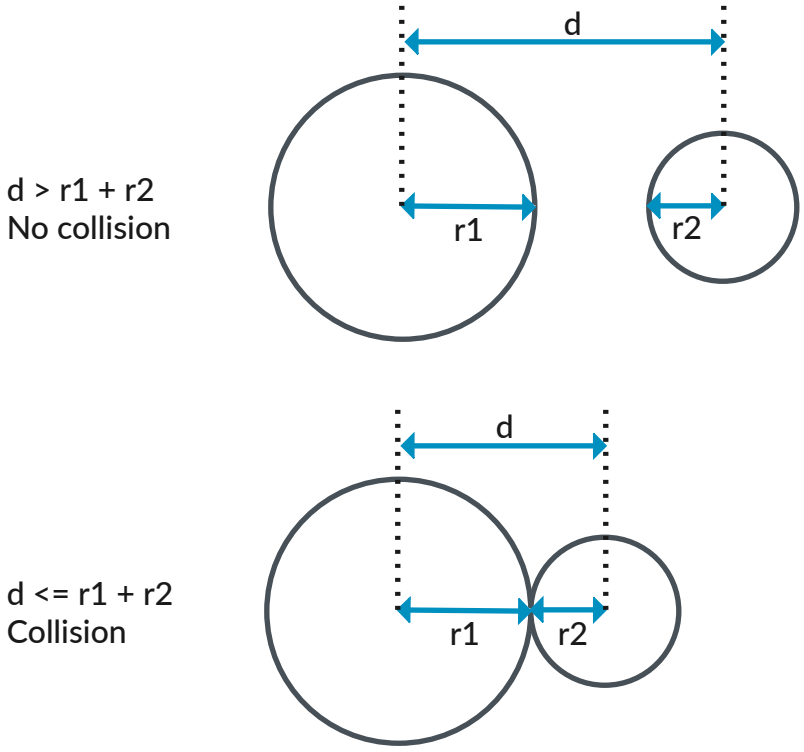
Collision detection algorithms let game software recognize when objects touch or collide.

Simple collision detection algorithm

To decide whether two objects have collided or not, we can simplify their shape to circles.

If we first consider collision detection along a single axis, we can see that collisions occur when these circles overlap. The following diagram shows this:

Figure 6-1: Collision detection on a single axis



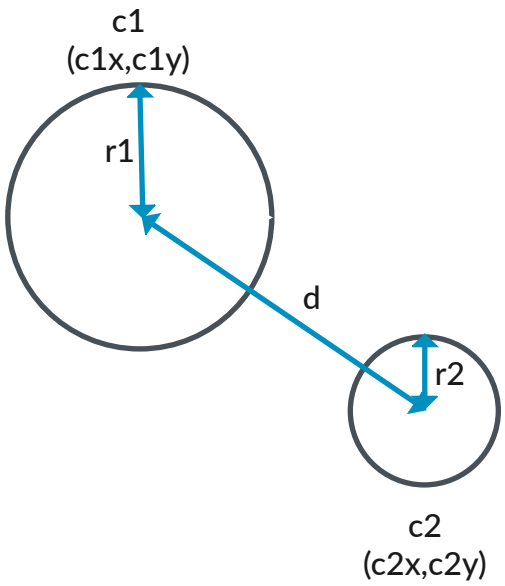
When one circle has radius r1, and another circle has radius r2:

- Circles collide when the distance between the center of the circles, d, is greater than the sum of r1 and r2.

- Circles do not collide when the distance between the center of the circles, d, is less than or equal to the sum of r1 and r2.

In two dimensions, each object has a pair of (x,y) coordinates specifying the center of the circle. Consider the same two circles, c1 and c2, at positions (c1x,c1y) and (c2x,c2y) respectively, as the following diagram shows:

Figure 6-2: Collision detection in two dimensions

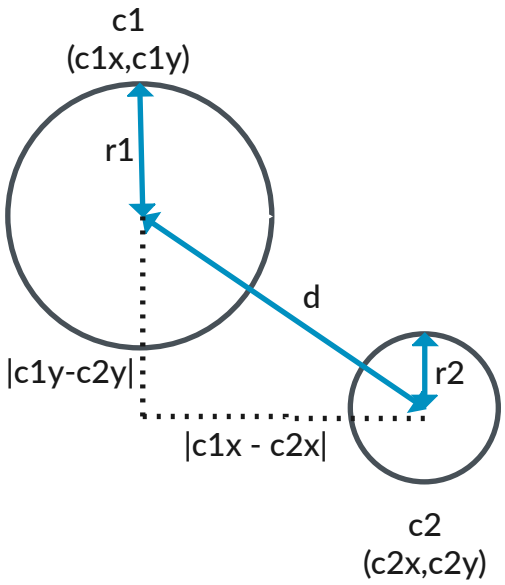


Collision detection in two dimensions is the same as on a single axis:

- Circles collide when the distance between the center of the circles, d, is greater than the sum of r1 and r2.
- Circles do not collide when the distance between the center of the circles, d, is less than or equal to the sum of r1 and r2.

To calculate the distance between the centers of the circles, imagine a right-angle triangle superimposed on the position of the circles as the following diagram shows:

Figure 6-3: Using Pythagoras' theorem to calculate the distance between circles



- The height of the triangle is the absolute difference between the y coordinates of the circles, | c1y - c2y|
- The base of the triangle is the absolute difference between the x coordinates of the circles, |c1x - c2x|
- The hypotenuse of the triangle, d, is the distance between the centers of the circles C1 and C2.

To calculate the hypotenuse, d, we use Pythagoras' theorem. This states that the square on the hypotenuse is equal to the sum of the squares on the other two sides:

$d^2 = (c1y - c2y)^2 + (c1x - c2x)^2$

Note

We can save ourselves having to calculate the square root, which can be a relatively expensive operation, by calculating d^2 instead of d. d^2 works as well as d for our collision detection algorithm: if d is less than the sum of the radii, then d^2 is also less than the sum of their squares. Using the squared values also means we do not need to calculate absolute values for the x and y differences, because squared values are always positive.

Algorithm implementation without vectorization

The collision detection algorithm takes the center points of two circles and checks the distance between the center points. If this distance is less than the sum of their radii, the circles have collided.

The following example shows the code without vectorization:

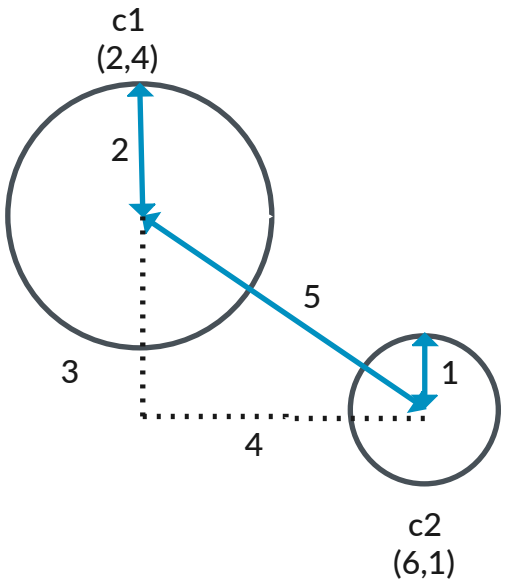
```
#include <stdio.h>

struct circle
{
  float radius;
  float x;
  float y;
};

bool does_collide(circle& c1, circle& c2)
{
  // Two circles collide if the distance from c1 to c2 is less
  // than the sum of their radii, or equivalently if the squared
  // distance is less than the square of the sum of the radii.
  float delta_x = c1.x - c2.x;
  float delta_y = c1.y - c2.y;
  float deltas_squared = (delta_x * delta_x) + (delta_y * delta_y);
  float radius_sum_squared = (c1.radius * c1.radius) + (c2.radius * c2.radius);
  return deltas_squared <= radius_sum_squared;
}

int main()
{
  circle c1;
  c1.radius = 2.0;
  c1.x = 2.0;
  c1.y = 4.0;

  circle c2;
  c2.radius = 1.0;
  c2.x = 6.0;
  c2.y = 1.0;

  if (does_collide(c1, c2)) {
    printf("Circles collide\n");
  } else {
    printf("Circles do not collide\n");
  }
  return (0);
}
```

This code creates two circles, as follows:

- c1, with radius 2 at coordinates (2,4)
- c2, with radius 1 at coordinates (6,1)

The difference between the x coordinates is 4 (6-2), and the difference between the y coordinates is 3 (4-1). This means the distance between the centers of the circles is 5 ($3^2 + 4^2 = 5^2$). The following diagram shows this calculation:

Figure 6-4: Example collision detection



The circles do not collide because the distance between the circles (5) is greater than the sum of the radii (3).

Compiling the C code with no vectorization produces the following assembly code:

```
ldr   s0, [x0]      // Load c1.x (first data at c1 base address)
ldr   s1, [x1]      // Load c2.x (first data at c2 base address)
fsub  s0, s0, s1    // Calculate (c1.x - c2.x)
ldr   s2, [x0, 4]   // Load c1.y (offset 4 from c1 base address)
ldr   s1, [x1, 4]   // Load c2.y (offset 4 from c2 base address)
fsub  s2, s2, s1    // Calculate (c1.y - c2.y)
ldr   s1, [x0, 8]   // Load c1.radius (offset 8 from c1 base address)
ldr   s3, [x1, 8]   // Load c2.radius (offset 8 from c2 base address)
fmul  s0, s0, s0    // Calculate (c1.x - c2.x)^2
fmul  s2, s2, s2    // Calculate (c1.y - c2.y)^2
fadd  s0, s0, s2    // Calculate ((c1.x - c2.x)^2 + (c1.y - c2.y)^2 )
fmul  s1, s1, s1    // Calculate c1.radius^2
fmul  s3, s3, s3    // Calculate c2.radius^2
fadd  s1, s1, s3    // Calculate (c1.radius^2 + c2.radius^2)
fcmpe s0, s1        // Test ((c1.x - c2.x)^2 + (c1.y - c2.y)^2 ) against
                    //   (c1.radius^2 + c2.radius^2)
cset  w0, mi        // If the result is negative (mi),
                    //   set the result (w0) to 1
ret
```

Basic vectorization using Neon intrinsics

Neon instructions perform the same calculation on multiple data values simultaneously.

Our collision detection algorithm contains several instances where the same mathematical operation is performed on different data values. These operations are candidates for optimization using Neon intrinsics:

- Subtraction:
  - We subtract c2.x from c1.x to calculate the difference in the x coordinates.
  - We subtract c2.y from c1.x to calculate the difference in the y coordinates.
- Multiplication:
  - We multiply c1.radius by itself to calculate the square.
  - We multiply c2.radius by itself to calculate the square.

The following code uses Neon intrinsics to optimize the collision detection algorithm:

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

struct circle
{
  float x;
  float y;
  float radius;
} __attribute__((aligned (64)));

bool does_collide_neon(circle const& c1, circle const& c2)
{
  float32x2_t c1_coords = vld1_f32(&c1.x);
  float32x2_t c2_coords = vld1_f32(&c2.x);

  float32x2_t deltas = vsub_f32(c1_coords, c2_coords);
  float32x2_t deltas_squared = vmul_f32(deltas, deltas);

  float sum_deltas_squared = vpadds_f32(deltas_squared);

  float radius_sum = c1.radius + c2.radius;
  float radius_sum_squared = radius_sum * radius_sum;
  return sum_deltas_squared <= radius_sum_squared;
}

int main()
{
  circle c1;
  c1.radius = 2.0;
  c1.x = 2.0;
  c1.y = 4.0;

  circle c2;
  c2.radius = 1.0;
  c2.x = 6.0;
  c2.y = 1.0;

  if (does_collide_neon(c1, c2)) {
    printf("Circles collide\n");
  } else {
    printf("Circles do not collide\n");
  }
  return (0);
}
```

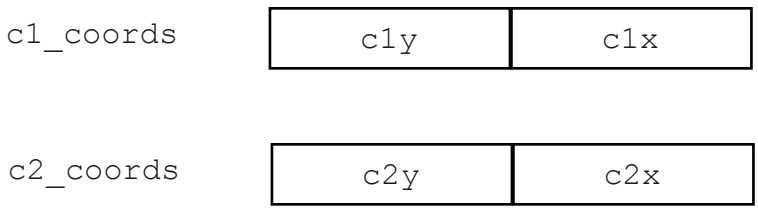
This code uses Neon intrinsics with 2-lane 64-bit Neon registers, each containing 32-bit floating-point values to parallelize parts of the algorithm. The portions of the code that use Neon intrinsics are as follows:

1. Load the x and y coordinates for each of the circles into different lanes of two separate Neon registers:

```
float32x2_t c1_coords = vld1_f32(&c1.x);
float32x2_t c2_coords = vld1_f32(&c2.x);
```

The `vld1_f32` intrinsic loads two 32-bit floating-point values into the Neon register, as the following diagram shows:

Figure 6-5: vld1_f32

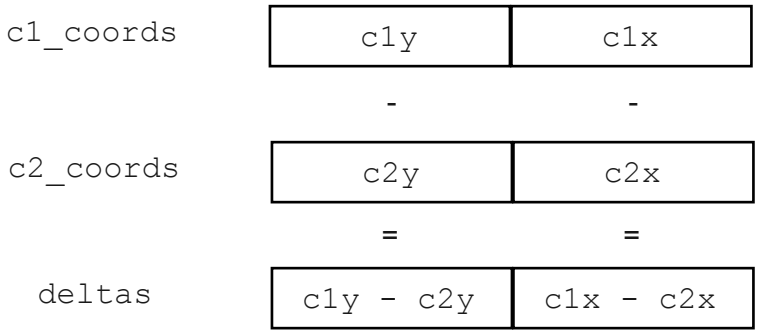


2. Subtract to obtain the deltas:

```
float32x2_t deltas = vsub_f32(c1_coords, c2_coords);
```

The `vsub_f32` intrinsic subtracts the vector elements in the second Neon register from the corresponding elements in the vector in the first register, and then places each result into elements of a destination register. The following diagram shows this:

Figure 6-6: vsub_f32

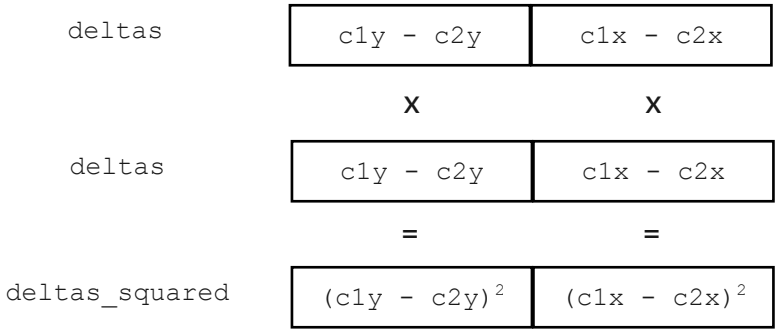


3. Multiply the deltas with themselves to obtain the square:

```
float32x2_t deltas_squared = vmul_f32(deltas, deltas);
```

The `vmul_f32` multiplies the 32-bit floating-point vector elements in the first Neon register by the values in the second source, and places the results in a vector. In our example, we use the same values to obtain the square of the deltas, as the following diagram shows:

Figure 6-7: vmul_f32



4. Sum across the vector to obtain a single scalar value, the square of the distance between the circles:

```
float sum_deltas_squared = vpadds_f32(deltas_squared);
```

The `vpadds_f32` adds the two floating-point vector elements in the Neon register to give a single scalar result.

Compiling the C code produces the following assembly code:

```
ldr   d0, [x0]            // Load c1.x and c1.y into two lanes of vector d0
                          //   first two data values at c1 base address
ldr   d1, [x1]            // Load c2.x and c2.y into two lanes of vector d1
                          //   first two data values at c2 base address
fsub  v0.2s, v0.2s, v1.2s // Vector subtract gets x and y deltas
fmul  v0.2s, v0.2s, v0.2s // Square both deltas at the same time
faddp s0, v0.2s           // Add across vector to get sum of squares
ldr   s1, [x0, 8]         // Code from here is scalar, same as before...
ldr   s2, [x1, 8]
fadd  s1, s1, s2
fmul  s1, s1, s1
fcmpe s0, s1
cset  w0, mi
ret
```

This code vectorizes the implementation of the algorithm by parallelizing the subtraction and multiplication operations when computing the square of the distance.

Advanced vectorization using Neon intrinsics

The solution presented in Basic vectorization using Neon intrinsics may not be faster than the original unvectorized implementation.

The performance gain from parallelizing the subtraction and multiplication operations is limited by the memory layout of our input. Loading the vector registers from this memory layout requires several instructions. Also, we only parallelized two operations, a subtraction and a multiplication, before needing to perform a cross-lane addition operation.

The declaration of the `circle` structure means that the data is interleaved, which inhibits vectorization.

An alternative approach is to change our data layout to provide deinterleaved data.

The following example shows this approach. It creates a new structure `circles` to hold pointers to arrays containing deinterleaved x, y, and radius data. This approach lets us check for collisions between a single `collider` circle and multiple `input` circles.

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

// Structure containing data for a single collider circle
struct circle
{
  float x;
  float y;
  float radius;
};

// Structure containing an array of pointers to data for multiple circles
struct circles
{
  size_t size;
```

```
  float* xs;
  float* ys;
  float* radii;
};

void does_collide_neon_deinterleaved(circles const& input, circle& collider, bool*
 out)
{
  // Duplicate the collider properties in 3 separate 4-lane vector registers
  float32x4_t c1_x = vdupq_n_f32(collider.x);
  float32x4_t c1_y = vdupq_n_f32(collider.y);
  float32x4_t c1_r = vdupq_n_f32(collider.radius);

  for (size_t offset = 0; offset != input.size; offset += 4)
  {
    // Perform 4 collision tests at a time
    float32x4_t x = vld1q_f32(input.xs + offset);
    float32x4_t y = vld1q_f32(input.ys + offset);

    float32x4_t delta_x = vsubq_f32(c1_x, x);
    float32x4_t delta_y = vsubq_f32(c1_y, y);
    float32x4_t delta_x_squared = vmulq_f32(delta_x, delta_x);
    float32x4_t delta_y_squared = vmulq_f32(delta_y, delta_y);
    float32x4_t sum_deltas_squared = vaddq_f32(delta_x_squared, delta_y_squared);

    float32x4_t r = vld1q_f32(input.radii + offset);
    float32x4_t radius_sum = vaddq_f32(c1_r, r);
    float32x4_t radius_sum_squared = vmulq_f32(radius_sum, radius_sum);
    uint32x4_t mask = vcltq_f32(sum_deltas_squared, radius_sum_squared);

    // Unpack the results in each lane
    out[offset] = 1 & vgetq_lane_u32(mask, 0);
    out[offset + 1] = 1 & vgetq_lane_u32(mask, 1);
    out[offset + 2] = 1 & vgetq_lane_u32(mask, 2);
    out[offset + 3] = 1 & vgetq_lane_u32(mask, 3);
  }
}

int main()
{
  int num_input = 4;
  float input_x[num_input] __attribute__((aligned (64)));
  float input_y[num_input] __attribute__((aligned (64)));
  float input_r[num_input] __attribute__((aligned (64)));
  bool output[num_input]   __attribute__((aligned (64)));

  // Set up the data for multiple circles
  for (int i = 0; i < num_input; i++) {
    input_x[i] = i*2;
    input_y[i] = i*3;
    input_r[i] = i;
    output[i] = 0;
  }

  // Create input object containing pointers to array data for multiple circles
  circles c1;
  c1.size = num_input;
  c1.radii = input_r;
  c1.xs = input_x;
  c1.ys = input_y;

  // Create collider object containing data for a single circle
  circle c2;
  c2.radius = 5.0;
  c2.x = 10.0;
  c2.y = 10.0;

  // Test whether the collider circle collides with any of the input circles,
 returning results in output
  does_collide_neon_deinterleaved(c1, c2, output);
```

```
  // Iterate over the returned output data and display results
  for (int i = 0; i < num_input; i++) {
    if (output[i]) {
      printf("Circle %d at (%.1f, %.1f) with radius %.1f collides\n", i, input_x[i],
 input_y[i], input_r[i]);
    } else {
    printf("Circle %d at (%.1f, %.1f) with radius %.1f does not collide\n", i,
 input_x[i], input_y[i], input_r[i]);
    }
  }
  return (0);
}
```

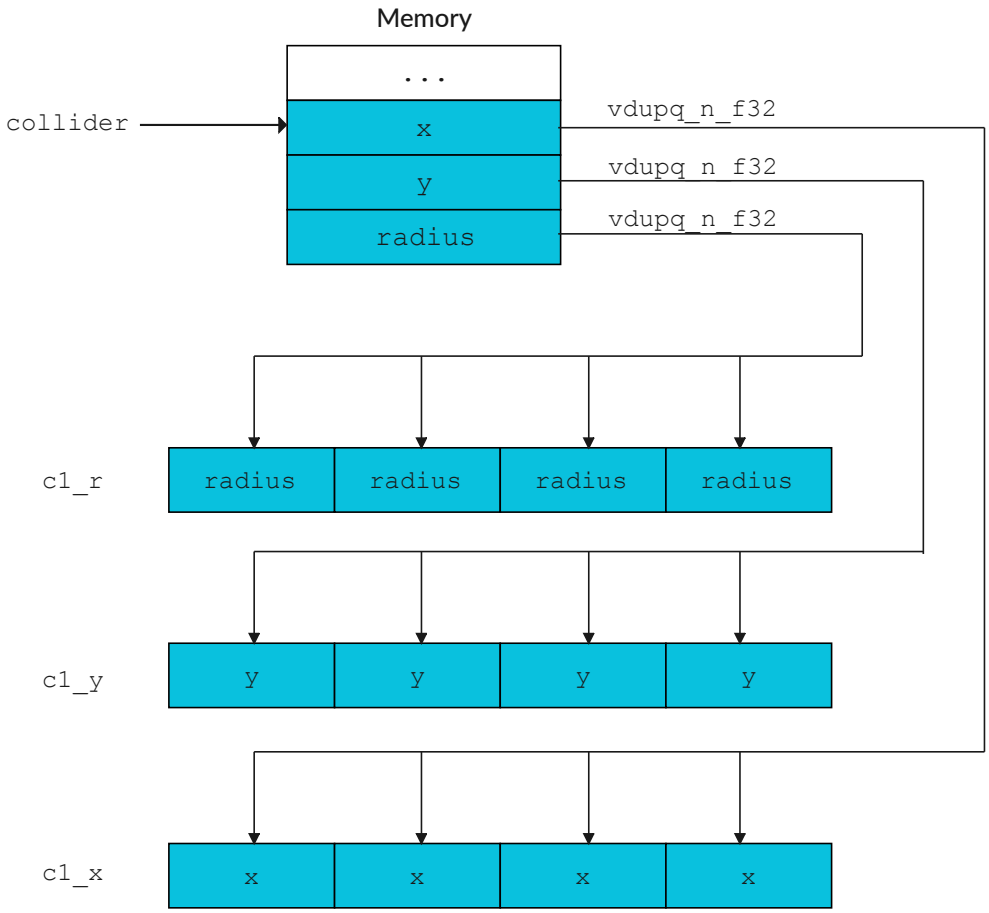
The portions of the code that use Neon intrinsics are as follows:

1. Duplicate the collider x, y, and radius properties into three separate Neon registers:

   ```
   float32x4_t c1_x = vdupq_n_f32(collider.x);
   float32x4_t c1_y = vdupq_n_f32(collider.y);
   float32x4_t c1_r = vdupq_n_f32(collider.radius);
   ```

   The `vdupq_n_f32` intrinsic duplicates a single scalar value into all the lanes of a Neon register, as the following diagram shows:

Figure 6-8: vdupq_n_f32



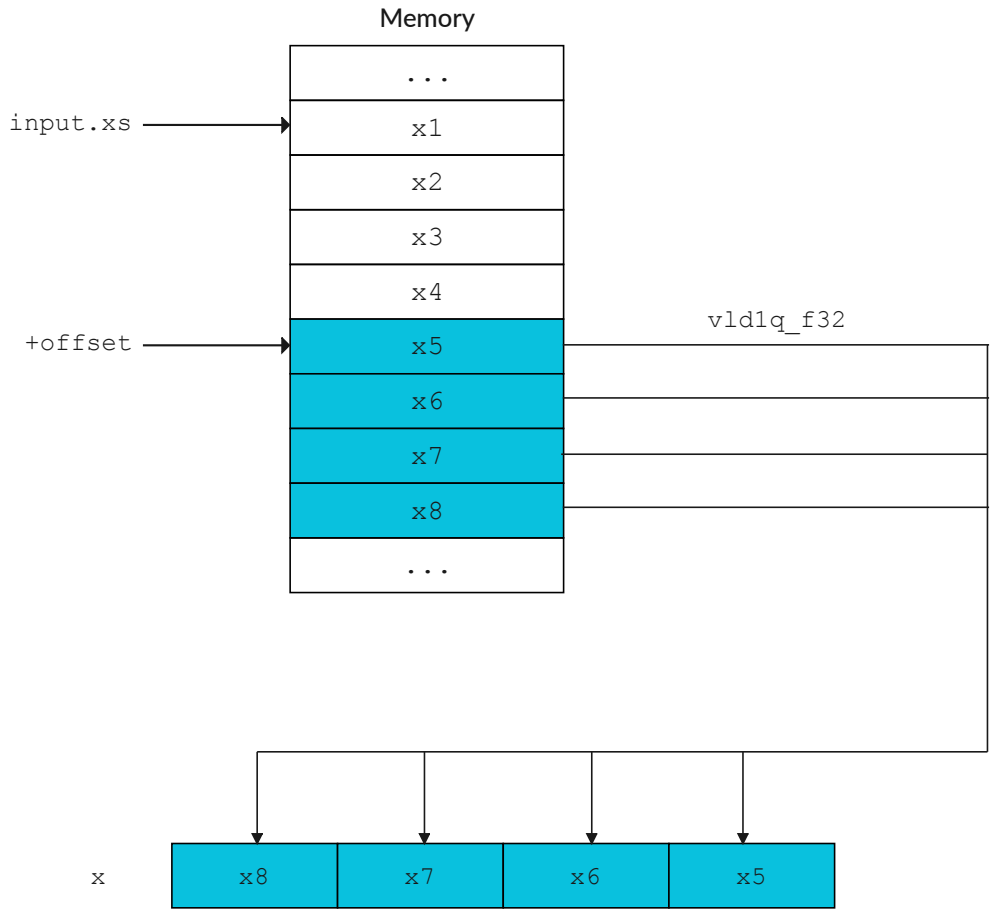
2. Load the x and y data for the next four input circles into separate lanes of two different Neon registers:

```
float32x4_t x = vld1q_f32(input.xs + offset);
float32x4_t y = vld1q_f32(input.ys + offset);
```

The `vld1q_f32` intrinsic loads four 32-bit values from consecutive memory addresses into four 32-bit lanes of a Neon register. The address is calculated by adding an offset to the base address pointer for both the x and y data. The offset increments by four on each loop iteration, because each loop deals with four circles.

The following diagram shows the second iteration of the loop, using an offset of 4:

Figure 6-9: vld1q_f32

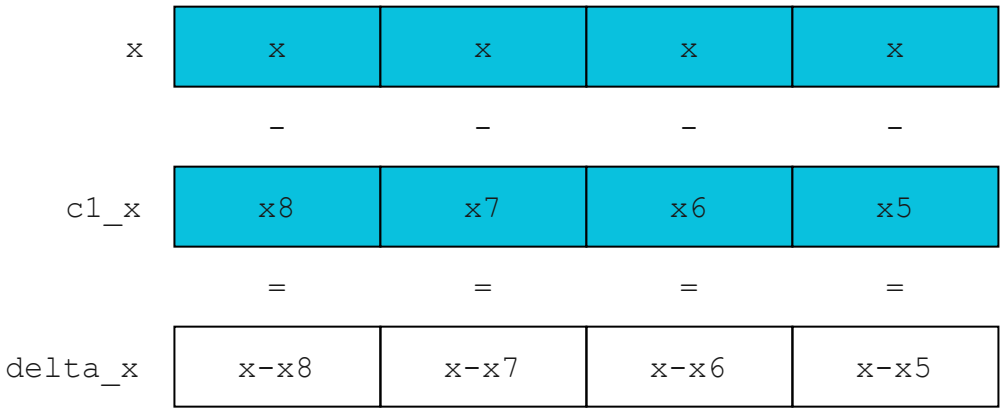


3. Subtract the collider `c1_x` value from each of the four input `x` values to obtain the `delta_x` value, and similar for `delta_y`:

```
float32x4_t delta_x = vsubq_f32(c1_x, x);
float32x4_t delta_y = vsubq_f32(c1_y, y);
```

The following diagram shows this:

Figure 6-10: vsubq_f32



4. Multiply the `delta_x` and `delta_y` values with themselves to obtain the squares:

```
float32x4_t delta_x_squared = vmulq_f32(delta_x, delta_x);
float32x4_t delta_y_squared = vmulq_f32(delta_y, delta_y);
```

5. Sum the lanes in the `delta_x_squared` and `delta_y_squared` separately to obtain the square of the distance:

```
float32x4_t sum_deltas_squared = vaddq_f32(delta_x_squared, delta_y_squared);
```

6. Use a similar process to calculate the squares of the radii:

```
float32x4_t r = vld1q_f32(input.radii + offset);
float32x4_t radius_sum = vaddq_f32(c1_r, r);
float32x4_t radius_sum_squared = vmulq_f32(radius_sum, radius_sum);
```

7. Compare `radius_sum_squared` to `sum_deltas_squared` for each of the four circles.

```
uint32x4_t mask = vcltq_f32(sum_deltas_squared, radius_sum_squared);
```

   The `vcltq_f32` intrinsic compares each lane of the first value with the corresponding lane in the second value. If the first value is less than the second, the intrinsic sets every bit of the corresponding lane in the result to 1, or 0 otherwise.

8. Extract each lane from the result individually, and set the next value in the `out` array to either 1 or 0.

```
out[offset] = 1 & vgetq_lane_u32(mask, 0);
out[offset + 1] = 1 & vgetq_lane_u32(mask, 1);
out[offset + 2] = 1 & vgetq_lane_u32(mask, 2);
out[offset + 3] = 1 & vgetq_lane_u32(mask, 3);
```

Performance results

This section describes the performance results for the three different implementations described in this guide.

When profiling performance, all functions were decorated with the GCC attribute `noinline` to prevent auto-vectorization by the compiler. This lets us compare the performance gains arising solely from our use of Neon intrinsics.

The following table summarizes the performance results:

Function	Time-per-invocation	Speedup
does_collide	2.724 ns	1x
does_collide_neon	2.717 ns	1.003x
does_collide_neon_deinterleaved	0.925 ns	2.945x

For each test above, the function in the left column performed 16,384 collision tests over 100,000 trials to compute the time-per-invocation in the center column. In all cases, the code was compiled with `-O3` and run on a Samsung S20.

The `does_collide_neon` speedup is nominal. However, restructuring the data in `does_collide_neon_deinterleaved` gives an impressive performance speedup of nearly 3x.

7. Program conventions

Program conventions are a set of guidelines for a specific programming language.

Macros

In order to use the intrinsics the Advanced SIMD architecture must be supported, and some specific instructions may or may not be enabled in any case. When the following macros are defined and equal to 1, the corresponding features are available:

__ARM_NEON
: Advanced SIMD is supported by the compiler. Always 1 for AArch64.

__ARM_NEON_FP
: Neon floating-point operations are supported. Always 1 for AArch64.

__ARM_FEATURE_CRYPTO
: Crypto instructions are available. Cryptographic Neon intrinsics are therefore available.

__ARM_FEATURE_FMA
: The fused multiply-accumulate instructions are available. Neon intrinsics which use these are therefore available.

This list is not exhaustive and further macros are detailed in the Arm C Language Extensions document.

Types

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

- `baseW_t` scalar data types
- `baseWxL_t` vector data types
- `baseWxLxN_t` vector array data types

Where:

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

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

In our earlier code we encountered an example of all three:

- `uint8_t`

- `uint8x16_t`
- `uint8x16x3_t`

Functions

As per the Arm C Language Extensions, the function prototypes from arm_neon.h follow a common pattern. At the most general level this is:

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

Be wary that some of the letters and names are overloaded, but in the order above:

ret
: the return type of the function.

v
: short for `vector` and is present on all the intrinsics.

p
: indicates a pairwise operation. (`[value]` means `value` may be present).

q
: indicates a saturating operation (with the exception of `vqtb[l][x]` in AArch64 operations where the `q` indicates 128-bit index and result operands).

r
: indicates a rounding operation.

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

u
: indicates signed-to-unsigned saturation.

n
: indicates a narrowing operation.

q
: postfixing the name indicates an operation on 128-bit vectors.

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

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

_n
: indicates a scalar operand supplied as an argument.

_lane

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

type

the primary operand type in short form.

args

the function's arguments.

8. Check your knowledge

Read the following questions to check 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.

Which header file must you include in a C file in order to use the Neon intrinsics?

`arm_neon.h#include <arm_neon.h>` must appear before the use of any Neon intrinsics.

What does this function do? `int8x16_t vmulq_s8 (int8x16_t a, int8x16_t b)`

The `mul` in the function name is a hint that this intrinsic uses the `MUL` instruction. Based on the types of the arguments and return value, sixteen bytes of signed integers, we might guess this intrinsic maps to the instruction `MUL Vd.16B, Vn.16B, Vm.16B`. So this function multiplies corresponding elements of `a` and `b` and returns the result. Checking the definition shows this is indeed true.

The `deinterleave` function defined in this tutorial can only operate on blocks of sixteen 8 bit unsigned integers. If you had an array of `uint8_t` values that was not a multiple of sixteen in length, how might you account for this while changing the arrays, but not the function and changing the function, but not the arrays?

To change the arrays but not the function, pad the arrays with zeros. This would be the simplest option, but this padding may have to be accounted for in other functions. To changing the function but not the arrays, use the Neon `deinterleave` for every whole multiple of sixteen values and then use the C `deinterleave` for the remainder.

What do the data types `float64_t`, `poly64x2_t`, and `int8x8x3_t` represent?

`float64_t` is a scalar type which is a 64-bit floating-point type. `poly64x2_t` is a vector type of two 64-bit polynomial scalars. `int8x8x3_t` is a vector array type of three vectors of eight 8-bit signed integers.

9. Related information

Here are some resources related to material in this guide:

- Engineering specifications for the Neon intrinsics can be found in the Arm C Language Extensions (ACLE).
- The Neon Intrinsics Reference provides a searchable reference of the functions specified by the ACLE.
- The Architecture Exploration Tools let you investigate the Advanced SIMD instruction set.
- The Arm Architecture Reference Manual provides a complete specification of the Advanced SIMD instruction set.
- Arm Cortex-A Software Development software training courses are designed to help engineers working on new or existing Cortex-A system designs.