



Learn the architecture - Optimizing C code with Neon intrinsics

1.0

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102467_0100_01_en



Learn the architecture - Optimizing C code with Neon intrinsics

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	11 March 2022	Non-Confidential	Initial release

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

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.....	6
2. What is Neon?.....	7
3. Why Neon intrinsics.....	8
4. Example - RGB deinterleaving.....	9
5. Example - matrix multiplication.....	11
6. Program conventions.....	20
7. Check your knowledge.....	23
8. Related information.....	24

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 demonstrate 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 will 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](#). First we will look at a simplified image processing example and matrix multiplication. Then we will move on to a more general discussion about the intrinsics themselves.

3. Why 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:

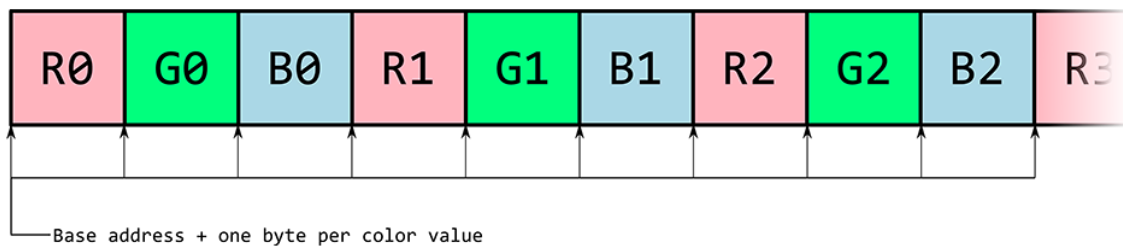
- There is a steeper learning curve to use Neon intrinsics than importing a library or relying on a compiler.
- Hand-optimized assembly code might offer the greatest scope for performance improvement even if it is more difficult to write.

We will 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 diagram:

Figure 4-1: RGB image pixel array



This is 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:

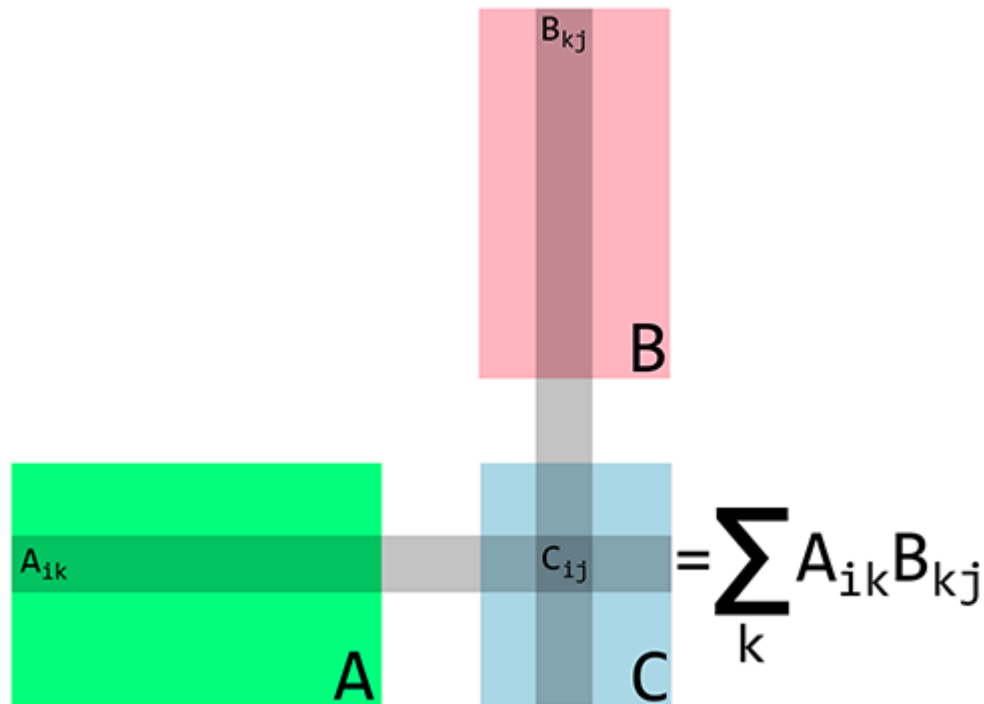
```
gcc -g -o3 rgb.c -o exe_rgb_o3
objdump -d exe_rgb_o3 > disasm_rgb_o3
```

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:

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 $n \times m$ matrix M is represented as an array M_array where $M_{ij} = M_array[n*j + i]$.

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 4×4 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 will make 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.

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 will enable 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,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);
    }
}
```

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

```
/*
 * 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. 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 `vqtbl[1][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.

7. 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 vmlq_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.

8. Related information

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.

Useful links to training

- [Introduction to Armv8-A](#)
- [Overview ISA](#)
- [Arm's other architectures](https://training.developer.arm.com/contents/406863)