Learn the architecture - Optimizing C code with Neon intrinsics

1.0

Non-Confidential

Issue 01
102467_0100_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

Proprietary Notice

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

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 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:

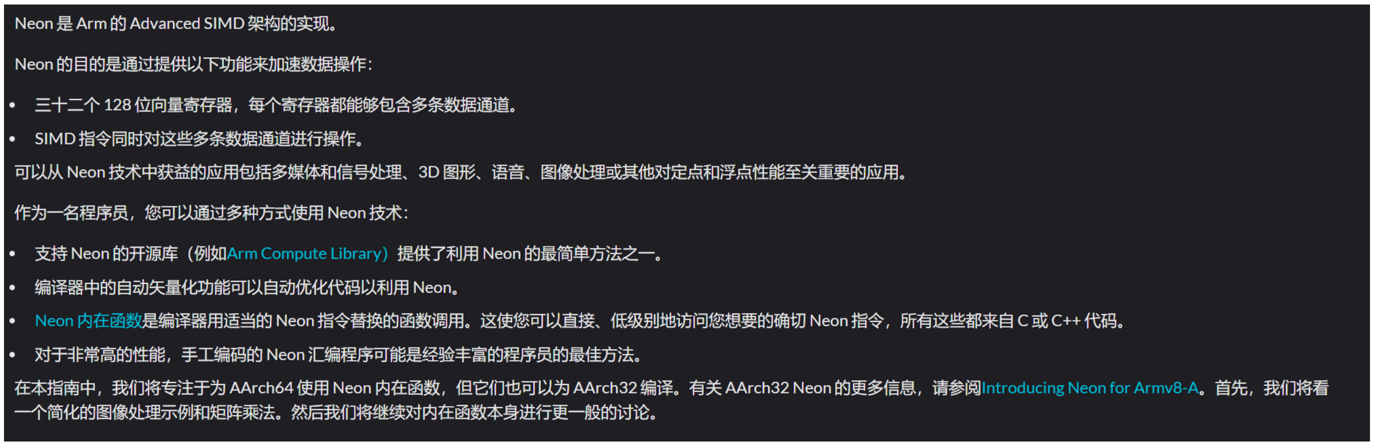
- Thirty-two 128-bit vector registers, each capable of containing multiple lanes of data.
- SIMD instructions to operate simultaneously on those multiple lanes of data.

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.



Neon 是 Arm 的 Advanced SIMD 架构的实现。

Neon 的目的是通过提供以下功能来加速数据操作：

- 三十二个 128 位向量寄存器，每个寄存器都能够包含多条数据通道。
- SIMD 指令同时对这些多条数据通道进行操作。

可以从 Neon 技术中获益的应用包括多媒体和信号处理、3D 图形、语音、图像处理或其他对定点和浮点性能至关重要的应用。

作为一名程序员，您可以通过多种方式使用 Neon 技术：

- 支持 Neon 的开源库（例如Arm Compute Library）提供了利用 Neon 的最简单方法之一。
- 编译器中的自动矢量化功能可以自动优化代码以利用 Neon。
- Neon 内在函数是编译器用适当的 Neon 指令替换的函数调用。这使您可以直接、低级别地访问您想要的确切 Neon 指令，所有这些都来自 C 或 C++ 代码。
- 对于非常高的性能，手工编码的 Neon 汇编程序可能是经验丰富的程序员的最佳方法。

在本指南中，我们将专注于为 AArch64 使用 Neon 内在函数，但它们也可以为 AArch32 编译。有关 AArch32 Neon 的更多信息，请参阅Introducing Neon for Armv8-A。首先，我们将看一个简化的图像处理示例和矩阵乘法。然后我们将继续对内在函数本身进行更一般的讨论。

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).

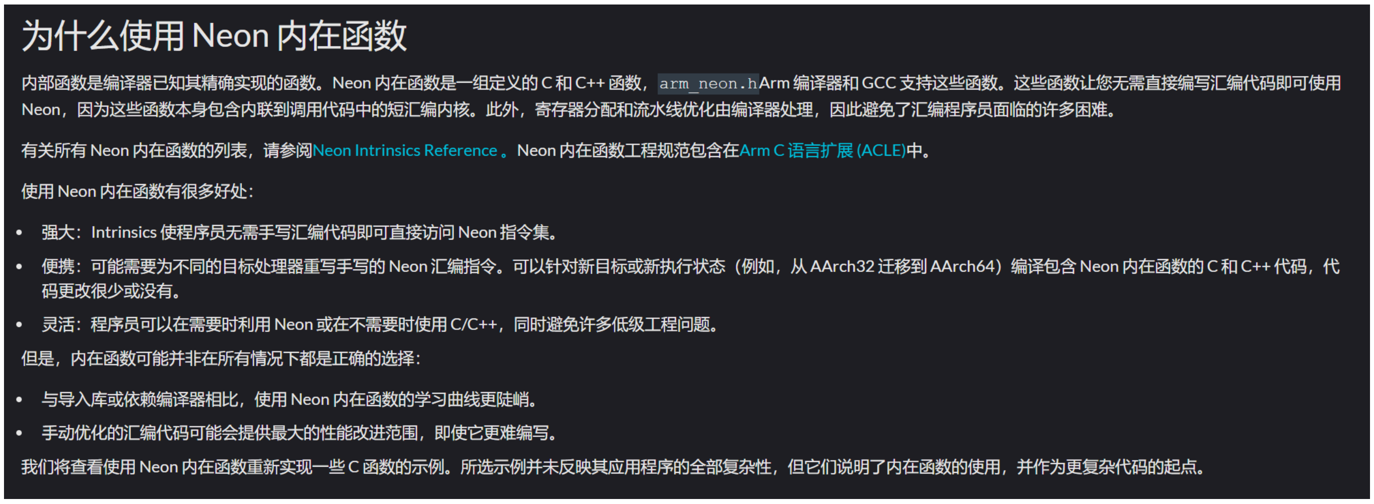
Using the Neon intrinsics has a number of benefits:

- 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.



为什么使用 Neon 内在函数

内部函数是编译器已知其精确实现的函数。Neon 内在函数是一组定义的 C 和 C++ 函数，`arm_neon.h`Arm 编译器和 GCC 支持这些函数。这些函数让您无需直接编写汇编代码即可使用 Neon，因为这些函数本身包含内联到调用代码中的短汇编内核。此外，寄存器分配和流水线优化由编译器处理，因此避免了汇编程序员面临的许多困难。

有关所有 Neon 内在函数的列表，请参阅Neon Intrinsics Reference 。Neon 内在函数工程规范包含在Arm C 语言扩展 (ACLE)中。

使用 Neon 内在函数有很多好处：

- 强大：Intrinsics 使程序员无需手写汇编代码即可直接访问 Neon 指令集。
- 便携：可能需要为不同的目标处理器重写手写的 Neon 汇编指令。可以针对新目标或新执行状态（例如，从 AArch32 迁移到 AArch64）编译包含 Neon 内在函数的 C 和 C++ 代码，代码更改很少或没有。
- 灵活：程序员可以在需要时利用 Neon 或在不需要时使用 C/C++，同时避免许多低级工程问题。

但是，内在函数可能并非在所有情况下都是正确的选择：

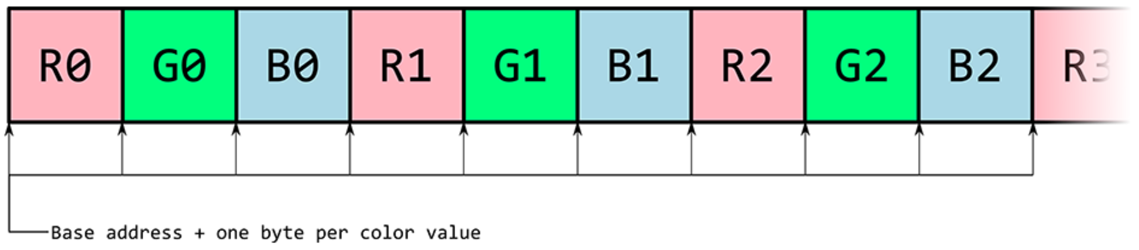
- 与导入库或依赖编译器相比，使用 Neon 内在函数的学习曲线更陡峭。
- 手动优化的汇编代码可能会提供最大的性能改进范围，即使它更难编写。

我们将查看使用 Neon 内在函数重新实现一些 C 函数的示例。所选示例并未反映其应用程序的全部复杂性，但它们说明了内在函数的使用，并作为更复杂代码的起点。

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.

这是因为 RGB 数据是交错的，访问和操作三个独立的颜色通道给程序员带来了问题。在简单的情况下，我们可以通过将模 3 应用于交错的 RGB 值来编写我们自己的单色通道操作。然而，对于更复杂的操作，例如傅里叶变换，提取和拆分通道会更有意义。

我们在内存中有一个 RGB 值数组，我们想要将它们解交错并将这些值放在单独的颜色数组中。执行此操作的 AC 程序可能如下所示：

```
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];
    }
}
```

但是有一个问题。在优化级别 -O3（非常高的优化）下使用 Arm Compiler 6 进行编译并检查反汇编显示没有使用 Neon 指令或寄存器。每个单独的 8 位值都存储在单独的 64 位通用寄存器中。考虑到全宽 Neon 寄存器为 128 位宽，在示例中每个寄存器可以容纳 16 个 8 位值，重写解决方案以使用 Neon 内部函数应该会给我们带来良好的结果。

```
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]);
    }
}
```

在此示例中，我们使用了以下类型和内在函数：

码元	它是什么?	我们为什么要使用它?
`uint8x16_t`	由 16 个 8 位无符号整数组成的数组。	一个`uint8x16_t`适合 128 位寄存器。即使在 C 代码中，我们也可以确保没有浪费的寄存器位。
`uint8x16x3_t`	具有三个元素的结构`uint8x16_t`。	循环中当前颜色值的临时保存区域。
`vld3q_u8(...)`	`uint8x16x3_t`一个通过加载 3*16 字节内存的连续区域返回 a 的函数。`uint8x16_t` 加载的每个字节都以交替模式放置在三个数组之一中。	在最低级别，此内在保证生成 LD3 指令，该指令以交替模式将给定地址的值加载到三个 Neon 寄存器中。
`vst1q_u8(...)`	`uint8x16_t`将 a 存储在给定地址的函数。	它存储一个充满字节值的完整 128 位寄存器。

可以使用以下命令在 Arm 机器上编译和反汇编上面的完整源代码：

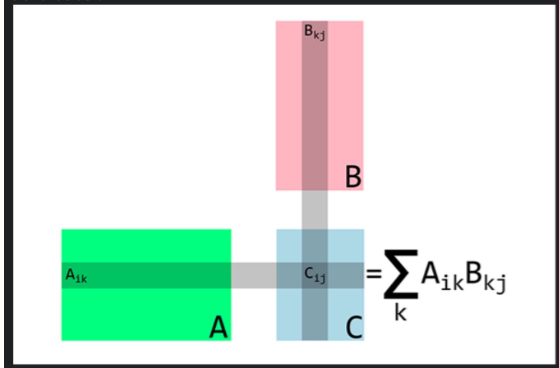
```
gcc -g -o3 rgb.c -o exe_rgb_o3
objdump -d exe_rgb_o3 > disasm_rgb_o3
```

如果您无法访问基于 Arm 的硬件，则可以使用Arm DS-5 Community Edition 和 Armv8-A Foundation Platform。

← 上一节　　　　下一节 →

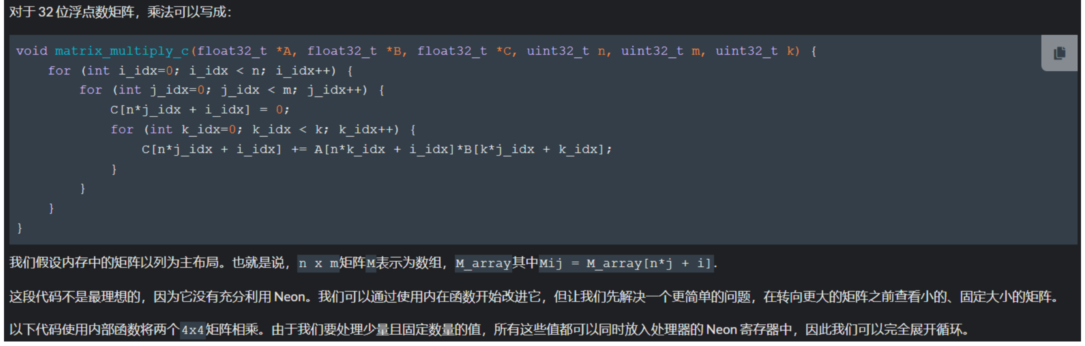
矩阵乘法是在许多数据密集型应用程序中执行的操作。它由一组以直接方式重复的算术运算组成：

图 1.矩阵乘法



矩阵乘法过程如下：

- A - 在第一个矩阵中取一行
- B - 执行此行与第二个矩阵中的列的点积
- C - 将结果存储在新矩阵的相应行和列中

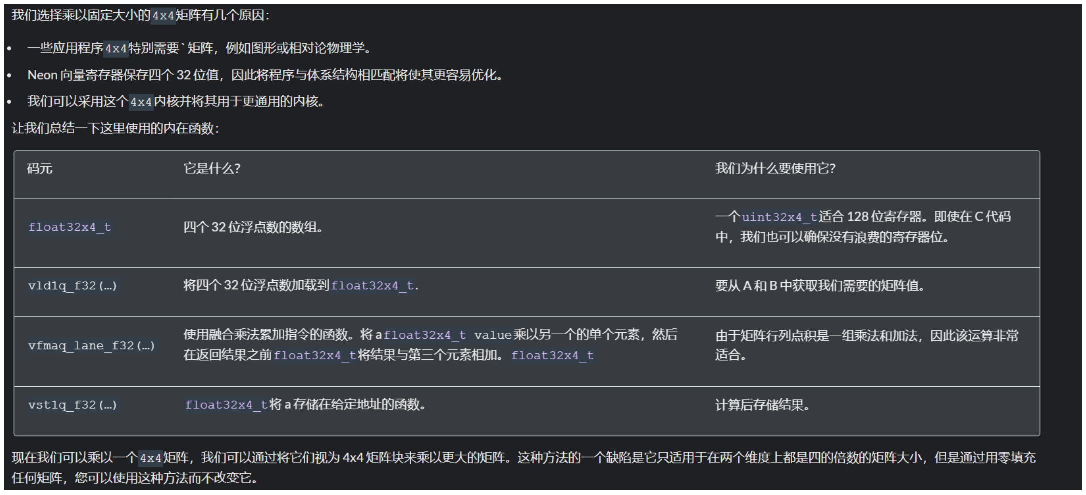

对于 32 位浮点数矩阵，乘法可以写成：

```
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];
            }
        }
    }
}
```

我们假设内存中的矩阵以列为主布局。也就是说，`n x m`矩阵`M`表示为数组，`M_array`其中`Mij = M_array[n*j + i]`.

这段代码不是最理想的，因为它没有充分利用 Neon。我们可以通过使用内在函数开始改进它，但让我们先解决一个更简单的问题，在转向更大的矩阵之前查看小的、固定大小的矩阵。

以下代码使用内部函数将两个`4x4`矩阵相乘。由于我们要处理少量且固定数量的值，所有这些值都可以同时放入处理器的 Neon 寄存器中，因此我们可以完全展开循环。

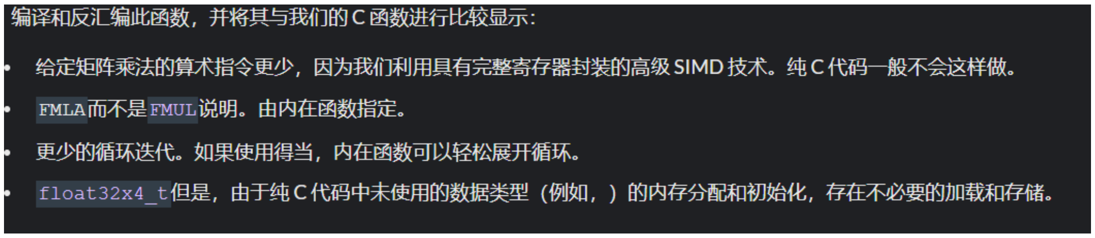

我们选择乘以固定大小的`4x4`矩阵有几个原因：

- 一些应用程序`4x4`特别需要`矩阵，例如图形或相对论物理学。
- Neon 向量寄存器保存四个 32 位值，因此将程序与体系结构相匹配将使其更容易优化。
- 我们可以采用这个`4x4`内核并将其用于更通用的内核。

让我们总结一下这里使用的内在函数：

码元	它是什么？	我们为什么要使用它？
`float32x4_t`	四个 32 位浮点数的数组。	一个`uint32x4_t`适合 128 位寄存器。即使在 C 代码中，我们也可以确保没有浪费的寄存器位。
`vld1q_f32` (...)	将四个 32 位浮点数加载到`float32x4_t`.	要从 A 和 B 中获取我们需要的矩阵值。
`vfmaq_lane_f32` (...)	使用融合乘法累加指令的函数。将 a`float32x4_t value`乘以另一个的单个元素，然后在返回结果之前`float32x4_t`将结果与第三个元素相加。`float32x4_t`	由于矩阵行列点积是一组乘法和加法，因此该运算非常适合。
`vst1q_f32` (...)	`float32x4_t`将 a 存储在给定地址的函数。	计算后存储结果。

现在我们可以乘以一个`4x4`矩阵，我们可以通过将它们视为 4x4 矩阵块来乘以更大的矩阵。这种方法的一个缺陷是它只适用于在两个维度上都是四的倍数的矩阵大小，但是通过用零填充任何矩阵，您可以使用这种方法而不改变它。

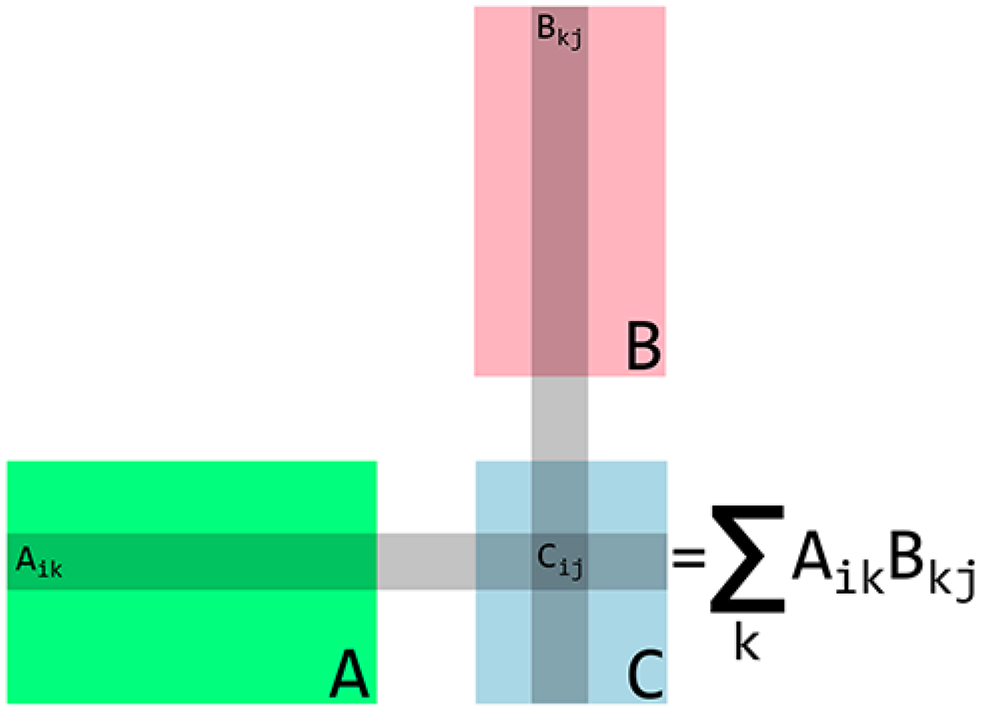

编译和反汇编此函数，并将其与我们的 C 函数进行比较显示：

- 给定矩阵乘法的算术指令更少，因为我们利用具有完整寄存器封装的高级 SIMD 技术。纯 C 代码一般不会这样做。
- `FMLA`而不是`FMUL`说明。由内在函数指定。
- 更少的循环迭代。如果使用得当，内在函数可以轻松展开循环。
- `float32x4_t`但是，由于纯 C 代码中未使用的数据类型（例如，）的内存分配和初始化，存在不必要的加载和存储。

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 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 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=v1d1q_f32(B+B_idx+k);
                      C1=vfmaq_laneq_f32(C1,A0,B1,0);
                      C1=vfmaq_laneq_f32(C1,A1,B1,1);
                      C1=vfmaq_laneq_f32(C1,A2,B1,2);
                      C1=vfmaq_laneq_f32(C1,A3,B1,3);

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

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

```
/*
 * 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 `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`

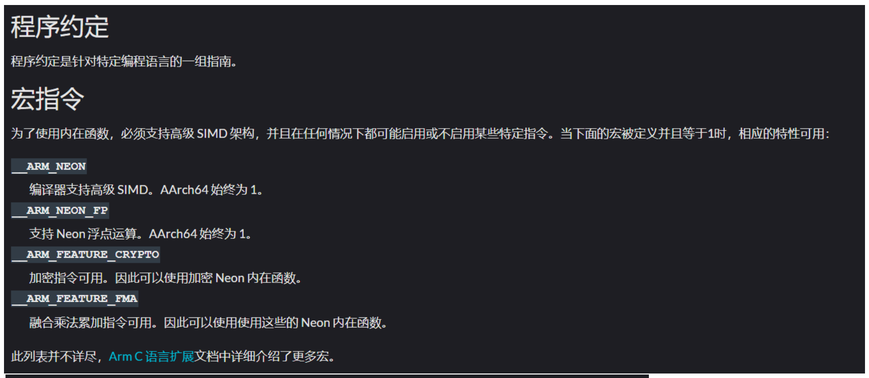
indicates a scalar operand taken from the lane of a vector. `_laneq` indicates a scalar operand taken from the lane of an input vector of 128-bit width. (`left | right` means only `left` or `right` would appear).

`type`

the primary operand type in short form.

`args`

the function's arguments.



程序约定

程序约定是针对特定编程语言的一组指南。

宏指令

为了使用内在函数，必须支持高级 SIMD 架构，并且在任何情况下都可能启用或不启用某些特定指令。当下面的宏被定义并且等于1时，相应的特性可用：

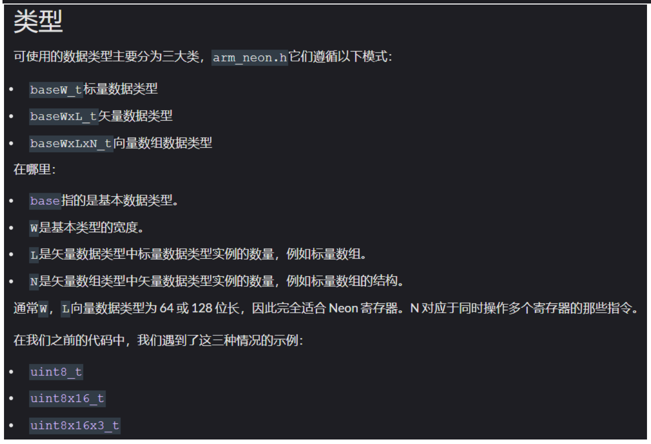
`__ARM_NEON`
编译器支持高级 SIMD。AArch64 始终为 1。

`__ARM_NEON_FP`
支持 Neon 浮点运算。AArch64 始终为 1。

`__ARM_FEATURE_CRYPTO`
加密指令可用。因此可以使用加密 Neon 内在函数。

`__ARM_FEATURE_FMA`
融合乘法累加指令可用。因此可以使用使用这些的 Neon 内在函数。

此列表并不详尽，Arm C 语言扩展文档中详细介绍了更多宏。



类型

可使用的数据类型主要分为三大类，`arm_neon.h`它们遵循以下模式：

- `baseW_t`标量数据类型
- `baseWxL_t`矢量数据类型
- `baseWxLxN_t`向量数组数据类型

在哪里：

- `base`指的是基本数据类型。
- `W`是基本类型的宽度。
- `L`是矢量数据类型中标量数据类型实例的数量，例如标量数组。
- `N`是矢量数组类型中矢量数据类型实例的数量，例如标量数组的结构。

通常`W`，`L`向量数据类型为 64 或 128 位长，因此完全适合 Neon 寄存器。N 对应于同时操作多个寄存器的那些指令。

在我们之前的代码中，我们遇到了这三种情况的示例：

- `uint8_t`
- `uint8x16_t`
- `uint8x16x3_t`

ret	函数的返回类型。
v	所有内在函数的缩写 vector，并且存在于所有内在函数中。
p	表示成对操作。 （手段 可能存在） 。 [value]value
q	vqtb[l][x]指示饱和操作 （ AArch64 操作 除外， 其中q指示 128 位索引和结果操作数） 。
r	表示舍入操作。
name	基本操作的描述性名称。这通常是高级 SIMD 指令，但并非必须如此。
u	表示有符号到无符号饱和度。
n	表示缩小操作。
q	名称后缀表示对 128 位向量的操作。
x	指示 AArch64 中的高级 SIMD 标量运算。b它可以是、 h或 s （ d即 8、16、32 或 64 位） 之一 。
_high	在 AArch64 中，用于涉及 128 位操作数的扩大和缩小操作。对于扩展 128 位操作数， high指的是源操作数的前 64 位。对于缩小，它指的是目标操作数的前 64 位。
_n	指示作为参数提供的标量操作数。
_lane	指示取自向量通道的标量操作数。 _laneq指示取自 128 位宽度的输入向量的通道的标量操作数。 （ left | right表示仅 left或 right将出现） 。
type	简写形式的主要操作数类型。
args	函数的参数。

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 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.

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)