

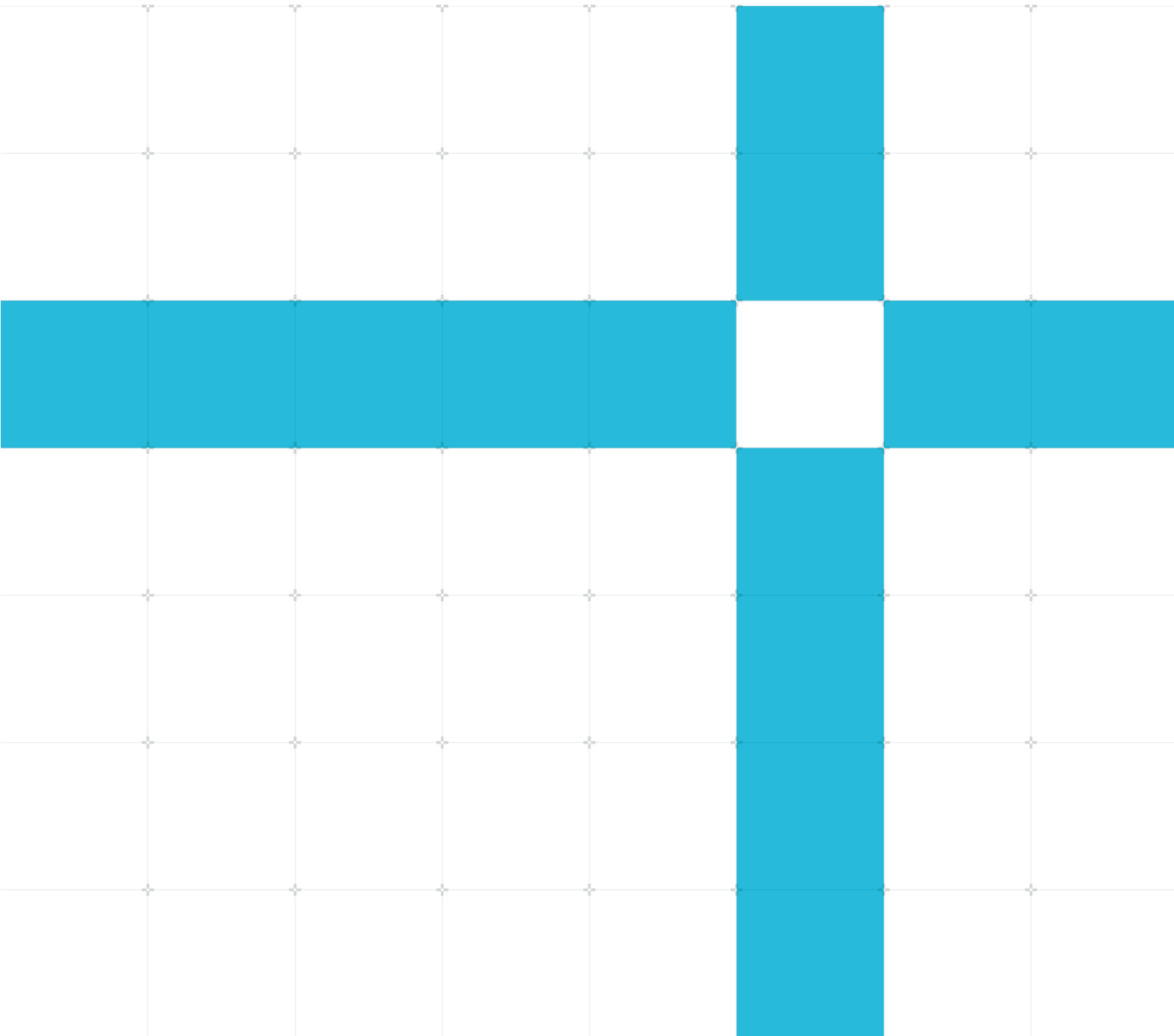


# Port Intel intrinsics to Neon technology

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

Issue 1

102581



## Neon technology

### Port Intel intrinsics to Neon technology

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

#### Release information

#### Document history

Issue	Date	Confidentiality	Change
01	August 3, 2021	Non-confidential	First release
02	October 18, 2021	Non-confidential	Port with SSE2Neon and SIMDDe section added

## Non-Confidential 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.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 <http://www.arm.com/company/policies/trademarks>.

Copyright © 2021 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)

## Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

## Product Status

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

## Web Address

[www.arm.com](http://www.arm.com)

# Contents

<b>1 Overview .....</b>	<b>5</b>
1.1 Before you begin.....	5
<b>2 About intrinsics .....</b>	<b>6</b>
2.1 How is Neon is different to SSE intrinsics? .....	6
<b>3 Porting intrinsics.....</b>	<b>9</b>
3.1 Port intrinsics manually.....	9
3.2 Port with platform-agnostic headers.....	14
3.3 Port to a unified vector library.....	18
<b>4 Port with SSE2Neon and SIMDDe.....</b>	<b>19</b>
4.1 What are the differences between SIMDDe and SSE2Neon? .....	21
4.2 Improve the code .....	22
<b>5 Related information .....</b>	<b>25</b>
<b>6 Next steps.....</b>	<b>26</b>

# 1 Overview

In this guide, you learn how to transition from x86 to Arm Neon technology for non-portable x86 Intel SSE code with sample code. This guide focuses on porting SSE intrinsics used on Intel and AMD hardware to Neon intrinsics with the Single Instruction Multiple Data (SIMD) instruction set.

This guide does not include the underlying assembly code that the intrinsics compile to, or the performance characteristics of your code after a port.

If you maintain code that is accelerated by SSE intrinsics on Intel and AMD platforms, you may have investigated how to port SSE code to Arm-powered devices. Previously, x86-targeted and Arm-targeted assembly code was partitioned along usage boundaries. x86 code typically ran in desktop and server environments, and Arm code ran on edge devices and mobile hardware.

With Windows running on Arm-based devices, it is increasingly important to support both x86 and Arm usage scenarios. Microsoft provides x86 emulation modes when running Windows operating systems on Arm. However, your program can suffer from reduced performance and thermal efficiency, compared to a native port.

## 1.1 Before you begin

To work through this guide, you need to be familiar with the Arm Cortex-A series processors and intrinsics. For more information about the Arm Cortex-A processor, read the [Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile](#). For an introduction to Neon intrinsics, you can read at explanation at [Why Neon intrinsics](#).

## 2 About intrinsics

In this section, you learn about intrinsics and how Neon intrinsics differ from SSE intrinsics. Intrinsics let the compiler assist the programmer. When code is expressed as intrinsics instead of raw assembly, the compiler is responsible for controlling register allocation. The compiler is also responsible for negotiating call conventions when traversing function call boundaries and can optimize the generated code.

SSE intrinsics look like the following code example:

```
#include <xmmintrin.h>

__m128 mul(__m128 a, __m128 b)
{
    return _mm_mul_ps(a, b);
}
```

This code defines a function `mul` which takes two 128-bit vectors as arguments, multiplies them lane-wise, and returns the result.

Compared to the SSE code example, the same function with Neon intrinsics looks like the following code:

```
#include <arm_neon.h>

float32x4_t mul(float32x4_t a, float32x4_t b)
{
    return vmulq_f32(a, b);
}
```

### 2.1 How is Neon is different to SSE intrinsics?

Neon intrinsics are different from SSE intrinsics in some important ways.

First, the specification of the input arguments and output result in Neon is a `float32x4_t` instead of a `__m128` type. Unlike SSE register types, Neon register types lead with the component type and are followed by the bit width of the component times the lane count.

For example, we want to port code that operates on 128-bit integers. The expected Neon type describes four 32-bit integers. The Neon register corresponding to `__m128i` in SSE is `int32x4_t`. The Neon type that corresponds to `__m128d` is contained in the 128-bit register. This register contains two 64-bit floats, and the Neon type is `float64x2_t`. SSE types describe the width of the entire vector register, and Neon types describe the width of each component and the component count.

Another important distinction between SSE and Neon types is the treatment of unsigned quantities. Neon encodes the signed nature of the data in the type itself by offering register types like `uint32x4_t` and `int32x4_t`. SSE offers only one register, `__m128i`, to store four 32-bit signed and unsigned integers.

SSE programmers who want the data to be treated as unsigned should choose the appropriate intrinsic function. Append the `_epu*` suffix to treat the operands as unsigned integral data. Neon enforces this data at the type level, and conversions are performed explicitly where necessary. There are fewer intrinsic function names for to memorize. This is because of the argument-dependent lookup.

If an overload is not supported, the compiler provides an error message, as shown in the following code:

```
#include <arm_neon.h>

uint32x4_t sat_add(uint32x4_t a, uint32x4_t b)
{
    return vqaddq_u32(a, b);
}

int32x4_t sat_add(int32x4_t a, int32x4_t b)
{
    // Compile error! "cannot convert 'int32x4_t' to 'uint32x4_t'"
    return vqaddq_u32(a, b);
}
```

This code uses the intrinsic `vqaddq_u32` to add unsigned integers in a vectorized method, saturating instead of overflowing. A64 GCC fails to compile the second function in this example. This is because `vqaddq_u32` is defined for unsigned types only.

Compared to reading SSE intrinsic functions, Neon functions have a learning curve. SSE intrinsics are typically structured as shown in the following code:

```
[width-prefix]_[op]_[return-type]
_mm_extract_epi32
```

For example, `_mm_extract_epi32` is an intrinsic operating on 128-bit registers, using the width prefix `_mm`. This intrinsic performs an extract operation to produce a 32-bit signed value. The intrinsic `_mm256_mul_ps` performs a `mul` operation on packed scalar floats in a 256-bit register.

In contrast, Neon intrinsics have the following form:

```
[op][q]_[type]
vaddq_f64
```

The `q` in the intrinsic name indicates that the intrinsic accepts 128-bit registers, as opposed to 64-bit registers. Many of the `op` names lead with a `v`, meaning vector.

For example, `vaddq_f64` performs a vector add of 64-bit floats. The `q` indicates that this intrinsic operates on 128-bit vectors. The accepted arguments must be `float64x2_t`, since only two 64-bit floats fit in a 128-bit vector.

Neon intrinsics also support operations that act on lanes of the SIMD register and other options. The full form of a Neon intrinsic and its specification is described in the Program conventions topic in

[Optimizing C Code with Neon Intrinsics](#). Read this topic to help decipher intrinsics when you encounter them, and to follow the [Porting intrinsics](#) section of this guide.



# 3 Porting intrinsics

In this guide, we learn about:

- [Porting SSE code to run on Arm processors manually](#)
- [Using platform-agnostic headers](#)
- [Porting to a unified vector library](#)
- [Porting with SSE2Neon and SIMDe](#)

## 3.1 Port intrinsics manually

When porting existing SSE code, you can manually port each SSE routine. Manual porting is suitable for porting short, isolated snippets of code. In addition, code with fewer rare intrinsics and wide registers that are 256-bit and greater is easier to port.

The following example code is from [Klein](#), a C++ library that is written using SSE intrinsics to compute operators in Geometric Algebra. The following SSE code conjugates a vector denoting the orientation of a plane with a rotor, also known as a quaternion, rotating the plane in space:

```
#include <xmmintrin.h>

#define KLN_SWIZZLE(reg, x, y, z, w) \
    _mm_shuffle_ps((reg), (reg), _MM_SHUFFLE(x, y, z, w))

// a := plane (components indicate orientation and distance from the origin)
// b := rotor (rotor group isomorphic to the quaternions)
__m128 rotate_plane(__m128 a, __m128 b) noexcept
{
    // LSB
    //
    // a0 (b2^2 + b1^2 + b0^2 + b3^2) e0 +
    //
    // (2a2(b0 b3 + b2 b1) +
    // 2a3(b1 b3 - b0 b2) +
    // a1 (b0^2 + b1^2 - b3^2 - b2^2)) e1 +
    //
    // (2a3(b0 b1 + b3 b2) +
    // 2a1(b2 b1 - b0 b3) +
    // a2 (b0^2 + b2^2 - b1^2 - b3^2)) e2 +
    //
    // (2a1(b0 b2 + b1 b3) +
    // 2a2(b3 b2 - b0 b1) +
```

```
// a3 (b0^2 + b3^2 - b2^2 - b1^2) e3
//
// MSB

// Double-cover scale
__m128 dc_scale = _mm_set_ps(2.f, 2.f, 2.f, 1.f);
__m128 b_xwyz = KLN_SWIZZLE(b, 2, 1, 3, 0);
__m128 b_xzwy = KLN_SWIZZLE(b, 1, 3, 2, 0);
__m128 b_xxxx = KLN_SWIZZLE(b, 0, 0, 0, 0);

__m128 tmp1
    = _mm_mul_ps(KLN_SWIZZLE(b, 0, 0, 0, 2), KLN_SWIZZLE(b, 2, 1, 3, 2));
tmp1 = _mm_add_ps(
    tmp1,
    _mm_mul_ps(KLN_SWIZZLE(b, 1, 3, 2, 1), KLN_SWIZZLE(b, 3, 2, 1, 1)));
// Scale later with (a0, a2, a3, a1)
tmp1 = _mm_mul_ps(tmp1, dc_scale);

__m128 tmp2 = _mm_mul_ps(b, b_xwyz);

tmp2 = _mm_sub_ps(tmp2,
    _mm_xor_ps(_mm_set_ss(-0.f),
        _mm_mul_ps(KLN_SWIZZLE(b, 0, 0, 0, 3),
            KLN_SWIZZLE(b, 1, 3, 2, 3))));
// Scale later with (a0, a3, a1, a2)
tmp2 = _mm_mul_ps(tmp2, dc_scale);

// Alternately add and subtract to improve low component stability
__m128 tmp3 = _mm_mul_ps(b, b);
tmp3 = _mm_sub_ps(tmp3, _mm_mul_ps(b_xwyz, b_xwyz));
tmp3 = _mm_add_ps(tmp3, _mm_mul_ps(b_xxxx, b_xxxx));
tmp3 = _mm_sub_ps(tmp3, _mm_mul_ps(b_xzwy, b_xzwy));
// Scale later with a

__m128 out = _mm_mul_ps(tmp1, KLN_SWIZZLE(a, 1, 3, 2, 0));
out = _mm_add_ps(out, _mm_mul_ps(tmp2, KLN_SWIZZLE(a, 2, 1, 3, 0)));
out = _mm_add_ps(out, _mm_mul_ps(tmp3, a));
return out;
}</xmmmintrin.h>
```

This code pattern will be familiar to SSE programmers. A general approach is to start from the component-by-component computation to be performed. In this case, we are given two four-component vectors as `__m128` registers. Then, we factor out common subexpressions in a vector method before composing and returning the result. The first parameter named `a` indicates a plane corresponding to the following implicit equation:

$$a_0 + a_1x + a_2y + a_3z = 0$$

**Figure 2: Vector equation**

The second parameter `b` is also a four-component register that represents the four components of a rotor. The operation we are computing is the sandwich operator, written as follows:

$$ba\tilde{b}$$

**Figure 1: Sandwich operator**

Let's start our port to Neon with the following function signature:

```
float32x4_t rotate_plane(float32x4_t a, float32x4_t b) noexcept
{
    // TODO
}
```

Next, we learn how to initialize a `float32x4_t` with constant values. Compilers allow us to specify initial values with standard aggregate initialization, as shown in the following code:

```
float32_t tmp[4] = {1.f, 2.f, 2.f, 2.f};
float32x4_t dc_scale = vld1q_f32(tmp);
```

In this code, the lowest address in the register comes first, unlike in the `_mm_set_ps` intrinsic, which leads with the most significant bytes first.

The swizzle operation in `_mm_shuffle_ps` is common in SSE code that is more difficult to port. This is because there is no mirroring intrinsic in Neon. To do this function, we need the following tools:

- [vgetq\\_lane\\_f32](#) to retrieve a specified component within a vector as a scalar. The corresponding intrinsic for setting a lane from a scalar is `vsetq_lane_f32`.
- [vcopyq\\_lane\\_f32](#) to move a component from one vector to another
- [vdupq\\_lane\\_f32](#) to broadcast a line to all four components

We can go line by line, replacing all swizzles with the corresponding lane queries and assignments.

Replacing the swizzles line by line is unlikely to produce good results on an Arm-powered device. For example, on Intel hardware, a shuffle has a one cycle latency penalty and throughput of one cycle per instruction. In contrast, the DUP instruction that is used to extract a lane has a three cycle penalty on an Arm Cortex-A78 processor. Each MOV that is needed to assign a lane incurs another two cycle latency penalty.

To get better performance with Neon, we need to get exposure to instructions that operate on more than a lane-by-lane granularity. For an overview of the options for data permutation, refer to the Permutation section in the [Neon Programmer's Guide for Armv8-A: Coding for Neon](#).

**vextq\_f32** extracts components from two separate vectors, combining them from a provided component index. In addition, there is a family of rev intrinsics which lets us reverse the order of components.

**Note:** To generate permutations, you can cast a float32x4\_t to a float64x2\_t or the reverse. Each REV16, REV32, or REV64 instruction has a two cycle latency penalty, but can combine many individual lane gets and sets.

After carefully permuting the input vectors minimally, we receive the following function:

```
#include <arm_neon.h>

float32x4_t rotate_plane(float32x4_t a, float32x4_t b) noexcept
{
    // LSB
    //
    // a0 (b0^2 + b1^2 + b2^2 + b3^2)) e0 + // tmp 4
    //
    // (2a2(b0 b3 + b2 b1) + // tmp 1
    // 2a3(b1 b3 - b0 b2) + // tmp 2
    // a1 (b0^2 + b1^2 - b3^2 - b2^2)) e1 + // tmp 3
    //
    // (2a3(b0 b1 + b3 b2) + // tmp 1
    // 2a1(b2 b1 - b0 b3) + // tmp 2
    // a2 (b0^2 + b2^2 - b1^2 - b3^2)) e2 + // tmp 3
    //
    // (2a1(b0 b2 + b1 b3) + // tmp 1
    // 2a2(b3 b2 - b0 b1) + // tmp 2
    // a3 (b0^2 + b3^2 - b2^2 - b1^2)) e3 // tmp 3
    //
    // MSB

    // Broadcast b[0] to all components of b_xxxx
    float32x4_t b_0000 = vdupq_laneq_f32(b, 0); // 3:1

    // Execution Latency : Execution Throughput in trailing comments

    // We need b_.312, b_.231, b_.123 (contents of component 0 don't matter)
    float32x4_t b_3012 = vextq_f32(b, b, 3); // 2:2
    float32x4_t b_3312 = vcopyq_laneq_f32(b_3012, 1, b, 3); // 2:2
```

```
float32x4_t b_1230 = vextq_f32(b, b, 1); // 2:2
float32x4_t b_1231 = vcopyq_laneq_f32(b_1230, 3, b, 1); // 2:2

// We also need a_.231 and a_.312
float32x4_t a_1230 = vextq_f32(a, a, 1); // 2:2
float32x4_t a_1231 = vcopyq_laneq_f32(a_1230, 3, a, 1); // 2:2
float32x4_t a_2311 = vextq_f32(a_1231, a_1231, 1); // 2:2
float32x4_t a_2312 = vcopyq_laneq_f32(a_2311, 3, a, 2); // 2:2

// After the permutations above are done, the rest of the port is more natural
float32x4_t tmp1 = vfmaq_f32(vmulq_f32(b_0000, b_3312), b_1231, b);
tmp1 = vmulq_f32(tmp1, a_1231);

float32x4_t tmp2 = vfmsq_f32(vmulq_f32(b, b_3312), b_0000, b_1231);
tmp2 = vmulq_f32(tmp2, a_2312);

float32x4_t tmp3_1 = vfmaq_f32(vmulq_f32(b_0000, b_0000), b, b);
float32x4_t tmp3_2 = vfmaq_f32(vmulq_f32(b_3312, b_3312), b_1231, b_1231);
float32x4_t tmp3 = vmulq_f32(vsubq_f32(tmp3_1, tmp3_2), a);

// tmp1 + tmp2 + tmp3
float32x4_t out = vaddq_f32(vaddq_f32(tmp1, tmp2), tmp3);

// Compute 0 component and set it directly
float32x4_t b2 = vmulq_f32(b, b);
// Add the top two components and the bottom two components
float32x2_t b2_hadd = vadd_f32(vget_high_f32(b2), vget_low_f32(b2));
// dot(b, b) in both float32 components
float32x2_t b_dot_b = vpadd_f32(b2_hadd, b2_hadd);

float32x4_t tmp4 = vmulq_lane_f32(a, b_dot_b, 0);
out = vcopyq_laneq_f32(out, 0, tmp4, 0);

return out;
}
```

The annotated expression in the comment at the top of the function shows how the temporaries that evaluate the expression are constructed. The compiled output code is a small routine of instructions, as shown in the following example:

```
rotate_plane(__Float32x4_t, __Float32x4_t):
ext v16.16b, v0.16b, v0.16b, #4
ext v3.16b, v1.16b, v1.16b, #12
```

```
mov v6.16b, v0.16b
fmul v4.4s, v1.4s, v1.4s
ins v16.s[3], v0.s[1]
ins v3.s[1], v1.s[3]
dup v2.4s, v1.s[0]
ext v7.16b, v1.16b, v1.16b, #4
ext v0.16b, v16.16b, v16.16b, #4
fmul v19.4s, v1.4s, v3.4s
fmul v18.4s, v2.4s, v3.4s
ins v7.s[3], v1.s[1]
ins v0.s[3], v6.s[2]
dup d17, v4.d[1]
dup d5, v4.d[0]
fmul v3.4s, v3.4s, v3.4s
mov v4.16b, v0.16b
mov v0.16b, v19.16b
fadd v5.2s, v5.2s, v17.2s
mov v17.16b, v18.16b
fmla v3.4s, v7.4s, v7.4s
fmls v0.4s, v2.4s, v7.4s
fmul v2.4s, v2.4s, v2.4s
faddp v5.2s, v5.2s, v5.2s
fmla v17.4s, v7.4s, v1.4s
fmul v0.4s, v4.4s, v0.4s
fmla v2.4s, v1.4s, v1.4s
fmul v5.4s, v6.4s, v5.s[0]
fmla v0.4s, v17.4s, v16.4s
fsub v2.4s, v2.4s, v3.4s
fmla v0.4s, v6.4s, v2.4s
ins v0.s[0], v5.s[0]
ret
```

With optimization settings set, Clang produces a better sequence of instructions to permute the vector. However, the optimizer might not notice the possible code improvements.

## 3.2 Port with platform-agnostic headers

The process of writing efficient intrinsics on Arm-powered hardware can seem complicated. Direct ports of SSE code to Neon can be time consuming, and do not always produce the wanted result.

The [SIMD Everywhere](#) (SIMDe) header-only library eases the task of porting. With SIMDe, the only change that your code needs is to replace the header, which is where you include platform intrinsics.

For example, you include the SIMDDe variant matching the instruction set that you originally targeted, instead of including `xmmintrin.h`.

SIMDe can be used to port x86 code to the Neon architecture. This custom code replaces the code that does not have a direct x86 to Neon functionality mapping. This code allows the source to benefit from performance optimization.

Internally, the SIMDDe header detects the target architecture that you are compiling to. The header also generates instructions that match the intrinsics that are used when writing code for the original target.

For example, in our original code, we had an `_mm_mul_ps` intrinsic. After changing the header to include the SIMDDe `sse.h` header, the code continues to invoke `_mm_mul_ps` when targeting x86 hardware. However, compiling for Neon also succeeds, because the SIMDDe header converts the `_mm_mul_ps` invocation to a `vmulq_f32`.

To see how this intrinsic rewriting is happening directly, refer to the [SIMDe implementation of `\_mm\_mul\_ps`](#). The same approach is taken for all supported intrinsics, and the SIMDDe implementation tries to select the most efficient replacement implementation possible. A commit like in this implementation can be sufficient to get set up with Neon quickly.

With a single line change to each file with SSE headers pointing to SIMDDe headers, you now have a codebase that can be compiled for Neon.

The next step is to profile the result to see if the performance of the SIMDDe direct replacement port is acceptable. Although porting with SIMDDe is quicker, direct replacement of x86 intrinsics with the Neon equivalents can result in inefficient code. By profiling the ported code, you can slowly migrate problematic portions of code to a native handwritten port on a case-by-case basis.

To see the effect of SIMDDe on our plane rotating function, we can swap out the line to include the SSE header with the following code:

```
#include <arm_neon.h>

typedef float32x4_t __m128;

inline __attribute__((always_inline)) __m128 _mm_set_ps(float e3, float e2, float e1,
float e0)
{
    __m128 r;
    alignas(16) float data[4] = {e0, e1, e2, e3};
    r = vld1q_f32(data);
    return r;
}

#define _MM_SHUFFLE(z, y, x, w) (((z) << 6) | ((y) << 4) | ((x) << 2) | (w))

inline __attribute__((always_inline)) __m128 _mm_mul_ps(__m128 a, __m128 b) {
    return vmulq_f32(a, b);
}
```

```

inline __attribute__((always_inline)) __m128 _mm_add_ps(__m128 a, __m128 b) {
    return vaddq_f32(a, b);
}

inline __attribute__((always_inline)) __m128 _mm_sub_ps(__m128 a, __m128 b) {
    return vaddq_f32(a, b);
}

inline __attribute__((always_inline)) __m128 _mm_set_ss(float a) {
    return vsetq_lane_f32(a, vdupq_n_f32(0.f), 0);
}

inline __attribute__((always_inline)) __m128 _mm_xor_ps(__m128 a, __m128 b) {
    return veorq_s32(a, b);
}

#define _mm_shuffle_ps(a, b, imm8) \
    __extension__({ \
        float32x4_t ret; \
        ret = vmovq_n_f32( \
            vgetq_lane_f32(a, (imm8) & (0x3))); \
        ret = vsetq_lane_f32( \
            vgetq_lane_f32(a, ((imm8) >> 2) & 0x3), \
            ret, 1); \
        ret = vsetq_lane_f32( \
            vgetq_lane_f32(b, ((imm8) >> 4) & 0x3), \
            ret, 2); \
        ret = vsetq_lane_f32( \
            vgetq_lane_f32(b, ((imm8) >> 6) & 0x3), \
            ret, 3); \
    })

```

These routines are lifted directly from the SIMD header. This means that you can see how the SSE intrinsics and shuffles map to Neon intrinsics. The following AArch64 assembly code is generated:

```

rotate_plane(__Float32x4_t, __Float32x4_t):    // @rotate_plane(__Float32x4_t,
__Float32x4_t)
    dup     v3.4s, v1.s[2]
    ext     v3.16b, v1.16b, v3.16b, #4
    dup     v2.4s, v1.s[0]
    ext     v20.16b, v1.16b, v3.16b, #12
    dup     v4.4s, v1.s[1]

```



```

dup      v5.4s, v1.s[3]
adrp     x8, .LCPI0_1
ext      v7.16b, v1.16b, v2.16b, #4
ext      v19.16b, v3.16b, v2.16b, #12
ext      v3.16b, v3.16b, v20.16b, #12
dup      v6.4s, v0.s[0]
ext      v16.16b, v1.16b, v4.16b, #4
ext      v5.16b, v1.16b, v5.16b, #4
ext      v17.16b, v1.16b, v7.16b, #12
ext      v18.16b, v1.16b, v7.16b, #8
fmul     v3.4s, v19.4s, v3.4s
ldr      q19, [x8, :lo12:.LCPI0_1]
ext      v6.16b, v0.16b, v6.16b, #4
ext      v17.16b, v7.16b, v17.16b, #12
ext      v7.16b, v7.16b, v18.16b, #12
ext      v18.16b, v1.16b, v16.16b, #8
ext      v20.16b, v1.16b, v5.16b, #8
ext      v2.16b, v5.16b, v2.16b, #12
ext      v16.16b, v16.16b, v18.16b, #12
ext      v18.16b, v0.16b, v6.16b, #8
ext      v5.16b, v5.16b, v20.16b, #12
ext      v20.16b, v0.16b, v6.16b, #12
adrp     x8, .LCPI0_0
ext      v18.16b, v6.16b, v18.16b, #12
ext      v6.16b, v6.16b, v20.16b, #12
fmul     v20.4s, v1.4s, v1.4s
fmul     v2.4s, v2.4s, v5.4s
fmul     v5.4s, v17.4s, v1.4s
mov      v1.s[0], v4.s[0]
ldr      q4, [x8, :lo12:.LCPI0_0]
eor      v2.16b, v2.16b, v19.16b
fmul     v1.4s, v16.4s, v1.4s
fadd     v2.4s, v5.4s, v2.4s
fmul     v5.4s, v17.4s, v17.4s
fadd     v5.4s, v20.4s, v5.4s
dup      v16.4s, v20.s[0]
fadd     v1.4s, v3.4s, v1.4s
fmul     v7.4s, v7.4s, v7.4s
fadd     v5.4s, v16.4s, v5.4s
fmul     v2.4s, v2.4s, v4.4s

```

```
fmul    v1.4s, v1.4s, v4.4s
fadd    v3.4s, v7.4s, v5.4s
fmul    v2.4s, v6.4s, v2.4s
fmul    v1.4s, v18.4s, v1.4s
fadd    v1.4s, v1.4s, v2.4s
fmul    v0.4s, v3.4s, v0.4s
fadd    v0.4s, v0.4s, v1.4s
ret
```

Even with the same -O2 optimization settings, the code contains 53 instructions with several DUP and Ext permutation intrinsics.

The effect of SIMDe on your codebase depends on several factors. One significant factor is the usage of SSE intrinsics that do not map well to Neon architecture.

### 3.3 Port to a unified vector library

You can use an intermediate library, for example [xsimd](#), to express vector manipulation and compilation.

Instead of maintaining a bespoke set of routines and algorithms for each instruction set, you use a common abstraction layer. This extraction layer has an efficient implementation on each supported architecture.

For smaller codebases, libraries like xsimd can be used to simplify working with vectorized code. Libraries can be useful if research and maintenance are needed to optimize bespoke implementations for each architecture.

The disadvantage to this approach is that integrating a library like xsimd is invasive. Optimization opportunities can be missed when you lose the capability to drop closer to the hardware. Sometimes, xsimd does not support certain operations, if they perform well on one architecture but poorly on another.

Despite these problems, using a library like xsimd can be better than using poor manual ports. For example, libraries can be useful for engineers who do not have the time to profile and optimize for each architecture.

## 4 Port with SSE2Neon and SIMDDe

Currently, games and applications are moving to devices that support Arm Neon. For example, PC and console games that have moved to mobile include Grand Theft Auto, Fortnite, and Brawlhalla. PC and laptop apps that support Arm Neon include Photoshop, Zoom, and Visual Studio Code.

When you move to Arm Neon, code in the app or game will need to be recompiled with the new target. If needed, there is help on [framework support](#), but what happens with handwritten Intel intrinsics code?

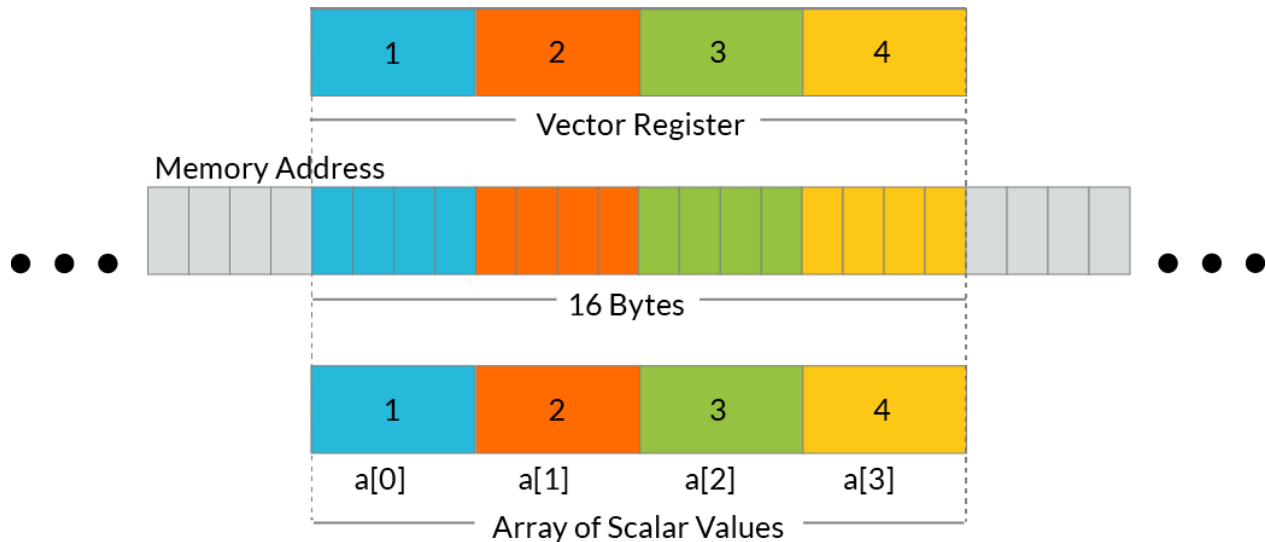
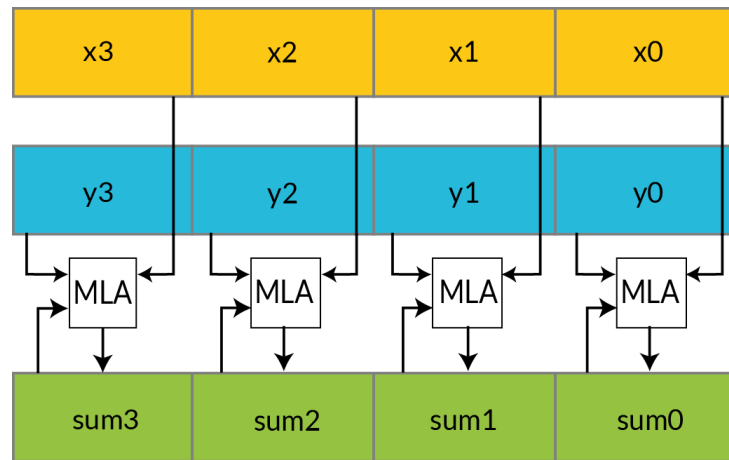


Figure 3: Intel memory to intrinsics register alignment



**Figure 4: Example intrinsics instruction Multiply Accumulate (MLA)**

Intrinsics provide an easier way to use specialized SIMD instructions than writing assembly code by hand. Intrinsics provide the same operations as assembly instructions, but directly in C/C++ and without needing to know which registers are used, the ABI, and so on.

Writing intrinsics code takes time, effort, and planning, especially if it is a new suite of commands you are unfamiliar with. An easier way is with one of the following open-source libraries:

- [SSE2Neon](#) converts Intel's SSE to Neon
- [SIMDe, or SIMD Everywhere](#) aims to make as much SIMD code translatable to as many different architectures as possible. Blender, OGRE, and FoundationDB are some products that have used one of these libraries to port to Arm Neon.

Intrinsics are used to optimize the performance of specific parts of the code. You need to decide if it is worth writing intrinsics code by hand to optimize efficiency, or if you would rather use the libraries to make porting and maintenance easier. This section helps you work out that balance.

In this section, we show how to port an SSE application to 64-bit Armv8-A architecture. For simplicity, this guide assumes we are porting from 128-bit SSE to 128-bit Neon. [Armv9-A](#) is similar for this Neon use case. You can also use this guide if you are converting 256-bit SSE. Although you would need to split everything in half to go from 256-bit SSE to 128-bit Neon, this split is also necessary if you are converting by hand.

Start your port with intrinsics because it simplifies the task of porting, whether you end up replacing library code with handwritten Neon.

The libraries allow you to replace as much or as little as you need with a true native port. For example, you could replace the library entirely and still use it in the porting process. The libraries allow you to quickly get your project compiling and working on Neon, even if you still want to improve performance. You do not need to rewrite all your SSE intrinsics code before anything works, and you have a functionally correct port working before you begin your optimization.

The libraries provide the best transliteration as possible, with a community contributing to the libraries to ensure good mapping without knowing why the intrinsics are called. However, when you are doing a port, you know the algorithm that calls the intrinsics and can adjust the surrounding code to suit Neon commands and get better improvements. Also, knowing your code and what you want to achieve, there may be Neon intrinsics available that are not in SSE that you could use. For example, in later Neon implementations there is complex number support with functions that can be done with one intrinsic call instead of many in an SSE function and its direct Neon translation. There are also areas like 8-bit support where many functions can be done directly with Neon calls, rather than in more complicated ways with SSE. In later Neon implementations and SSE implementations, there are different Dot Product intrinsics implementations, but the libraries fail to map them to each other. Instead, the libraries implement many multiply accumulates. In situations like this, you may be able to use these newer, more specialized Neon intrinsics to improve performance.

Generally, the first step for your SSE to Neon port is simply to get your application running on Neon hardware. If the intrinsics use of your project is very small, it might be most efficient to port by hand: you can quickly make SSE and Neon variants of the small number of functions, and you can consider the intrinsics available and adapt your algorithm around the intrinsics for efficiency. However, if your project uses intrinsics more extensively, porting by hand is harder. In this case, you might decide to use a library to help make porting easier.

## 4.1 What are the differences between SIMDDe and SSE2Neon?

If you decide to use a library, which library should you use?

The SIMDDe and SSE2Neon libraries both work in the same way. In your code, replace the `#include` for the Intel intrinsics header file with an `#include` for the library header file. In the implementation, the libraries detect what intrinsics are available from the compiler. The outcomes of this detection are as follows:

- Compiling for Intel uses the original SSE intrinsics implementation
- Compiling for Arm converts to Neon
- Compiling for a platform that is neither Intel or Arm uses a non-intrinsics implementation

For SSE2Neon, the `#include` changes are all that is needed.

For SIMDDe, there is a `#define` to avoid code changes beyond the different `#includes`, but we recommend that you make code changes by adding SIMDDe prefixes to the SSE functions for clarity.

SSE2Neon supports MMX and SSE, but if you have AVX code, use SIMDDe. SIMDDe supports AVX and AVX2, but only has partial support for AVX512. If you are considering a one-way port, use the simplicity of SSE2Neon rather than all the options that SIMDDe gives you.

SIMDDe offers ways to add further ports and future technology changes. If you want to support WebAssembly or implement a feature in Neon that automatically works on SSE, these options are covered in SIMDDe. SIMDDe intends to expand over time, so you will be able to use new SIMD technologies as they are released from both Arm and Intel. For example, SVE2 is an improvement on Neon in [Arm's v9 architecture](#).

SIMDe and SSE2Neon are both open source and benefit from contributions that improve functions to be more efficient. SSE2Neon is older and its code was initially used for SIMDe for the SSE to Neon use case, and the implementations are almost identical. Presently both libraries are maintained and there does not seem to be a performance reason to choose one over the other.

## 4.2 Improve the code

After you port with the library and have a functionally correct version of code running on Arm, you need to decide if your code can benefit from any improvements. Intrinsics increase performance in code but if mappings are not perfect from SSE to Neon, that code might need to be improved. There are some intrinsics that have perfect mapping and performance improvements might not be possible.

The Arm device may have different bottlenecks to an Intel device, and there may be different pieces of code that need to be improved. For the intrinsics, we look at which pieces of code are likely to require a closer look.

Most basic arithmetic and logic functions are either a perfect mapping or almost perfect. Along with the reinterpreted intrinsics being no-cost compiler directives, the load, set, and store intrinsics also map well with their library translations. Simple math will often not need any additional work. For example, a 2D distance calculation can be left to the library translation. The following code snippet shows a vectorized 2D distance calculation implemented using SSE intrinsics:

```
void distances(float* xDists, float* yDists, float* results, int size)
{
    __m128 Xs, Ys, m1, m2, m3, res;

    for (size_t index = 0; index < size; index += 4)
    {
        Xs = _mm_load_ps(xDists + index);
        Ys = _mm_load_ps(yDists + index);
        m1 = _mm_mul_ps(Xs, Xs);
        m2 = _mm_mul_ps(Ys, Ys);
        m3 = _mm_add_ps(m1, m2);
        res = _mm_sqrt_ps(m3);
        _mm_store_ps(results + index, res);
    }
}
```

The following code shows the resulting assembly code after GCC compilation with SIMDe Neon translation on [Godbolt](#):

```
distances(float*, float*, float*, int):
    sxtw    x4, w3
    cbz     w3, .L1
    sub     x3, x4, #1
    add     x4, x0, 16
    lsr     x3, x3, 2
```

```

        add    x3, x4, x3, lsl 4
.L3:
        ldr    q0, [x1], 16
        ldr    q1, [x0], 16
        fmul   v0.4s, v0.4s, v0.4s
        fmla   v0.4s, v1.4s, v1.4s
        fsqrt  v0.4s, v0.4s
        str    q0, [x2], 16
        cmp    x0, x3
        bne    .L3
.L1:
        ret

```

The loads and stores are not perfect because the SSE ordering of the vector in the SIMD type is the reverse of Neon, but this difference does not cause major issues.

Functions can be categorized as follows:

- Basic arithmetic and logic functions are either a perfect mapping or almost perfect. Loads, sets, and stores mostly map well.
- More complicated maths functions such as `sqrt`, `avg`, `min`, and `max` map well from SSE to Neon, although `abs` is usually not ideal but can sometimes be perfect.
- Bit shifts are an area where you may get a gain by writing some native Neon, negations, and aligns.
- Converts, inserts, and extracts generally map well and should be low priority for specialized Neon code.
- Compares mostly map well, although gains may be possible on specific compares to NaN, 0 or 1 where Intel has specialized functions.
- Moves mostly map well, but movemasks mostly do not.
- Higher precision division and `sqrt` make the mapping less efficient. Consider whether you need the precision.
- Pairwise functions are horizontal operations where the values within one vector are used together. Pairwise functions can be less efficient, especially subtractions. These functions have potential for improvement by specialized Neon code.
- Very specialized SSE functions like `_mm_dp_ps` and `_mm_minpos_epu16` could be very difficult to implement efficiently in Neon. Using a different algorithm for Neon could have a significant improvement.
- Crypto functions and bit tests may benefit from implementing specialized Neon code.
- Blends, shuffles, and rounding have more potential for gain than arithmetic-type functions.
- Narrowing and Widening have potential for gain with handwritten code.
- Recent function additions to Neon or SSE are less likely to be well mapped yet but may be an opportunity to contribute to the open-source libraries.

How much you replace will depend not just on efficiency, but on time available to port and the cost of maintenance. Having the SSE functions run on Neon without regard for performance requires almost no overhead, but optimizations made using specialized code changes need to be implemented and tested twice. These decisions about how much specialized code to write need to be made on a project-by-project basis.



## 5 Related information

Here are some resources related to the material in this guide:

- [Arm Performance Libraries](#)
- [Neon Intrinsics reference search engine](#)
- [Neon Programmer's Guide for Armv8-A: Coding for Neon](#)
- [SIMD Everywhere](#)
- [xsimd project on GitHub](#)

## 6 Next steps

In this guide, you learned that porting SSE code to Neon by hand is preferable if you do not have much code to port or if the performance needed is known to push hardware boundaries. For smaller codebases, libraries like xsimd can be used to simplify working with vectorized code. Finally, rather than writing or re-writing code to use an abstraction layer like xsimd, SIMDDe can be used to port x86 code to the architecture.

After your initial port and conducting benchmarks, we recommend that you refer to the [Arm Cortex-A78 Core Software Optimization Guide](#) for the specific chips you intend to target.