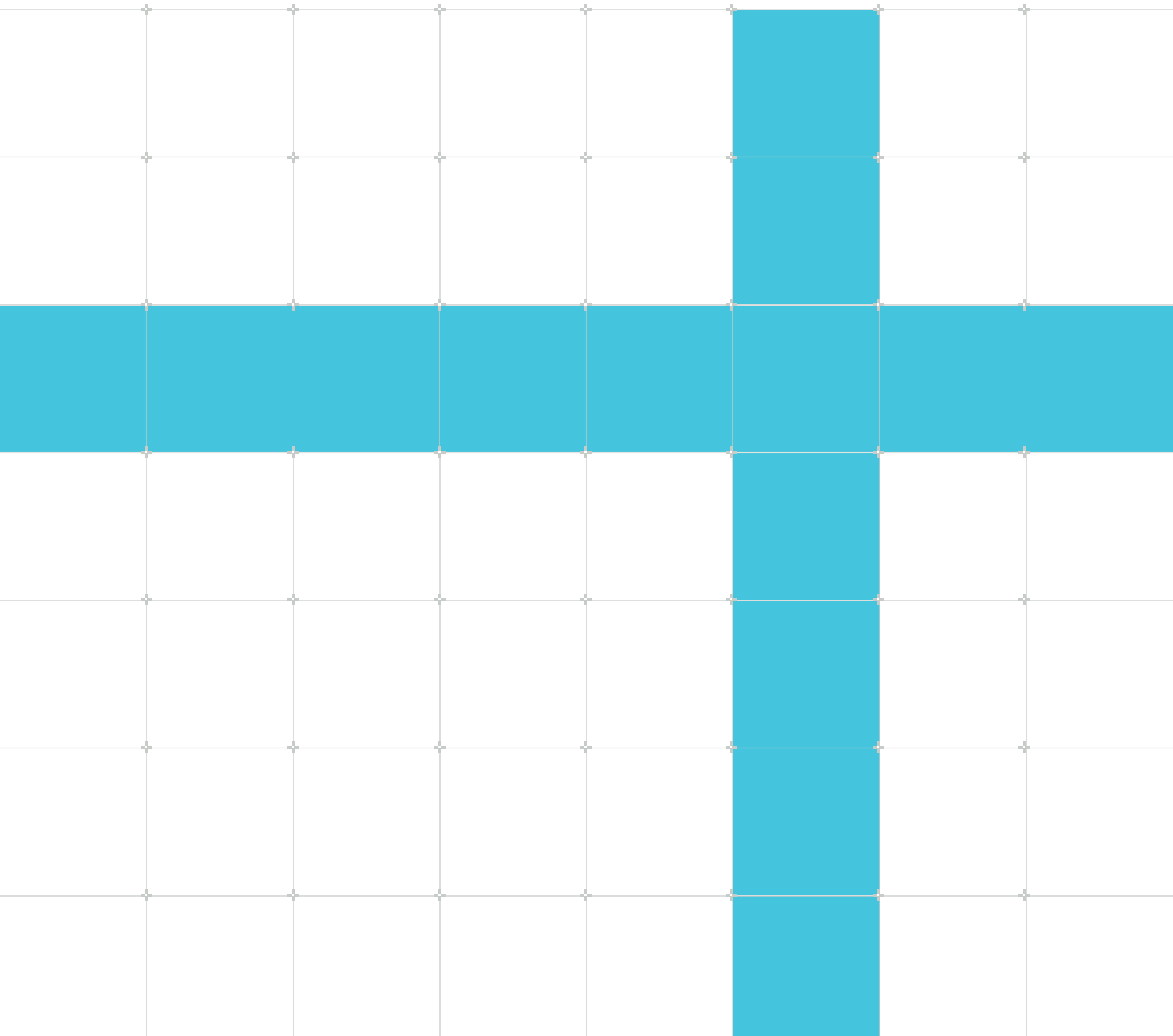
arm

Optimizing Image Processing with Neon Intrinsics

Version 3.0

Issue 01
101964_0300_01_en



Optimizing Image Processing with Neon Intrinsics

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	20 December 2019	Non-Confidential	First release
0200-00	8 July 2020	Non-Confidential	Updates to text.
0201-01	6 July 2021	Non-Confidential	Title update
0300-00	17 March 2023	Non-Confidential	Extended scope of guide to describe libTIFF optimizations.
0300-01	8 January 2024	Non-Confidential	Minor update

Proprietary Notice

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.

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 of Neon technology

This guide shows how Neon technology has been used to improve performance in the real-world: specifically, in the open-source Chromium and libTIFF projects.

In the guide, we demonstrate to programmers how they can use Neon intrinsics in their code to enable Single Instruction, Multiple Data (SIMD) processing. Using Neon in this way can bring huge performance benefits.

If you are not familiar with Neon, you can read an overview of Neon on the Arm Developer website.

What are Neon intrinsics?

Neon technology provides a dedicated extension to the Arm Instruction Set Architecture, providing additional instructions that can perform mathematical operations in parallel on multiple data streams.

Neon technology can help speed up a wide variety of applications, including:

- Audio and video processing
- 2D and 3D gaming graphics
- Voice and facial recognition
- Computer vision and deep learning

Neon intrinsics are function calls that programmers can use in their C or C++ code. The compiler then replaces these function calls with an appropriate Neon instruction or sequence of Neon instructions.

Intrinsics provide almost as much control as writing assembly language, but leave low-level details such as register allocation and instruction scheduling to the compiler. This frees developers to concentrate on the higher-level behavior of their algorithms, rather than the lower-level implementation details.

Another advantage of using intrinsics is that the same source code can be compiled for different targets. This means that, for example, a single source code implementation can be built for both 32-bit and 64-bit targets.

2. Chromium optimization with Neon intrinsics

This section of the guide examines several optimizations made to the Chromium open-source project using Neon intrinsics. These optimizations improve the performance of PNG image processing.

Why Chromium?

Why did we choose Chromium to investigate the performance improvements possible with Neon?

Chromium provides the basis for Google Chrome, which is the most popular web browser in the world, in terms of user numbers. Any performance improvements that we made to the Chromium codebase can benefit many millions of users worldwide.

Chromium is an open-source project, so everyone can inspect the full source code. When learning about a new subject, like programming with Neon intrinsics, it often helps to have examples to learn from. We hope that the examples that are provided in this guide help, because you can see them in the context of a complete, real-world codebase.

Why PNG?

Now that we have decided to work in Chromium, where should we look in the Chromium code to make optimizations? With over 25 million lines of code, we must pick a specific area to target. When looking at the type of workloads that web browsers deal with, the bulk of content is still text and graphics. Images often represent most of the downloaded bytes on a web page, and contribute to a significant proportion of the processing time. Recent data suggests that % of mobile users abandon sites that take over 3 seconds to load. This means that optimizing image load times, and therefore page load times, should bring tangible benefits.

The Portable Network Graphics (PNG) format was developed as an improved, non-patented replacement for the Graphics Interchange Format (GIF). PNG is the standard for transparent images in the web. It is also a popular format for web graphics in general. Because of this, Arm decided to investigate opportunities for Neon optimization in PNG image processing.

Introducing Bobby the bird

To help decide where to look for optimization opportunities, we went in search of performance data.

The following image of a bird has complex textures, a reasonably large size, and a transparent background. This means that it is a good test case for investigating optimizations to the PNG decoding process:

Figure 2-1: Bobby the bird

Image source: Penubag [Public domain], from Wikimedia Commons

The first thing to know is that all PNG images are not created equally. There are several different ways to encode PNG images, for example:

- Compression. Different compression algorithms can result in different file sizes. For example, Zopfli produces PNG image files that are typically about 5% smaller than zlib, at the cost of taking longer to perform the compression.
- Pre-compression filters. The PNG format allows filtering of the image data to improve compression results. PNG filters are lossless, so they do not affect the content of the image itself. Filters only change the data representation of the image to make it more compressible. Using pre-compression filters can give smaller file sizes at the cost of increased processing time.
- Color depth. Reducing the number of colors in an image reduces file size, but also potentially degrades image quality.
- Color indexing. The PNG format allows individual pixel colors to be specified as either a TrueColor RGB triple, or an index into a palette of colors. Indexing colors reduces file sizes, but might degrade image quality if the original image contains more colors than the maximum

that the palette allows. Indexed colors also need decoding back to the RGB triple, which may increase processing time.

We investigated performance with three different versions of the Bobby the budgie image to investigate possible areas for optimization.

Table 2-1: Performance using different images

Image	File size	Number of colors	Palette or TrueColor?	Filters?	Compression	Encoder
Original_Bobby.PNG	2.7M	211787	TrueColor	Yes	zlib	libpng
Palette_Bobby.PNG	0.9M	256	Palette	No	zlib	libpng
Zopfli_Bobby.PNG	2.6M	211787	TrueColor	Yes	Zopfli	ZopfliPNG

To obtain performance data for each of these three images, we used the Linux perf tool to profile ContentShell. The performance data for each image is as follows:

```
Original_Bobby.PNG
== Image has pre-compression filters (2.7MB) ==
Lib             Command SharedObj                       method
    CPU (%)
zlib            TileWorker      liblink
 inflate_fast..................... 1.96
zlib            TileWorker      libblnk
 adler32.......................... 0.88
blink  TileWorker       liblink                 ImageFrame::setRGBAPremultiply..
 0.45
blink   TileWorker      liblink
 png_read_filter_row_up...........0.03*
```

```
Palette_Bobby.PNG
== Image has no pre-compression filters (0.9MB) ==
Lib             Command SharedObj                       method
    CPU (%)
libpng  TileWorker      liblink                 cr_png_do_expand_palette........
 0.88
zlib            TileWorker      liblink
 inflate_fast..................... 0.62
blink   TileWorker      liblink                 ImageFrame::setRGBAPremultiply..
 0.49
zlib            TileWorker      libblnk
 adler32.......................... 0.31
```

```
Zopfli_Bobby.PNG
== Image was optimized using zopfli (2.6MB) ==
Lib             Command SharedObj                       method
    CPU (%)
zlib            TileWorker      liblink
 inflate_fast..................... 3.06
zlib            TileWorker      libblnk
 adler32.......................... 1.36
blink   TileWorker      liblink                 ImageFrame::setRGBAPremultiply..
 0.70
blink   TileWorker      liblink                 png_read_filter_row_up..........
 0.48*
```

This data helped identify the zlib library as a good target for our optimization efforts. This is because it contains several methods that contribute significantly to performance.

Zlib was also considered a good candidate to target for the following reasons:

- The zlib library is used in many different software applications and libraries, for example libpng, Skia, FreeType, Cronet, and Chrome. This means that any performance improvements that we could achieve in zlib would yield performance improvements for many users.
- Released in 1995, the zlib library has a relatively old codebase. Older codebases might have areas that have not been modified in many years. These areas are likely to provide more opportunities for improvement.
- The zlib library did not contain any existing optimizations for Arm. This means that there probably a wide range of improvements to make.

3. Chromium optimization: Adler-32

Adler-32 is a checksum algorithm used by the zlib compression library to detect data corruption errors. Adler-32 checksums are faster to calculate than CRC32 checksums, but trade reliability for speed as Adler-32 is more prone to collisions.

The PNG format uses Adler-32 for uncompressed data and CRC32 is used for the compressed segments.

An Adler-32 checksum is calculated as follows:

- A is a 16-bit checksum calculated as the sum of all bits in the input stream, plus 1, modulo 65521.
- B is a 16-bit checksum calculated as the sum of all individual A values, modulo 65521. B has the initial value 0.
- The final Adler-32 checksum is a 32-bit checksum formed by concatenating the two 16-bit values of A and B, with B occupying the most significant bytes.

This means that the Adler-32 checksum function can be expressed as follows:

```
A = 1 + D1 + D2 + ... + Dn (mod 65521)

B = (1 + D1) + (1 + D1 + D2) + ... + (1 + D1 + D2 + ... + Dn) (mod 65521)
  = nxD1 + (n-1) x D2 + (n-2) x D3 + ... + Dn + n (mod 65521)

Adler-32(D) = (B x 65536) + A
```

For example, the following table shows the calculation of the Adler-32 checksum of the ASCII string Neon:

Table 3-1: Adler-32 checksum calculation

Character	Decimal ASCII code	A	B
N	78	(1 + 78) % 65521 = 79	(0 + 79) % 65521 = 79
e	101	(79 + 101) % 65521 = 180	(79 + 180) % 65521 = 259
o	111	(180 + 111) % 65521 = 291	(259 + 291) % 65521 = 550
n	110	(291 + 110) % 65521 = 401	(550 + 401) % 65521 = 951

The decimal Adler-32 checksum is calculated as follows:

```
Adler-32 = (B x 65536) + A
         = (951 x 65536) + 401
         = 62,324,736 + 401
         = 62,325,137
```

The same calculation in hexadecimal is as follows:

```
Adler-32 = (B x 00010000) + A
         = (03B7 x 00010000) + 0191
         = 03B70000 + 0191
```

```
          = 03B70191
```

Unoptimized implementation

The following code shows a simplistic implementation of the Adler-32 algorithm, from Wikipedia:

```
const uint32_t MOD_ADLER = 65521;

uint32_t adler32(unsigned char *data, size_t len)
/*
    where data is the location of the data in physical memory and
    len is the length of the data in bytes
*/
{
    uint32_t a = 1, b = 0;
    size_t index;

    // Process each byte of the data in order
    for (index = 0; index < len; ++index)
    {
        a = (a + data[index]) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }

    return (b << 16) | a;
}
```

This code simply loops through the data one value at a time, summing and accumulating results. One of the problems with this approach is that performing the modulo operation is expensive. Here, this expensive modulo operation is performed at every single iteration.

Neon-optimized implementation

Optimizing the Adler-32 algorithm with Neon uses vector multiplication and accumulation to process up to 32 data values at the same time:

```
static void NEON_accum32(uint32_t *s, const unsigned char *buf,
                         z_size_t len)
{
    /* Please refer to the 'Algorithm' section of:
     * https://en.wikipedia.org/wiki/Adler-32
     * Here, 'taps' represents the 'n' scalar multiplier of 'B', which
     * will be multiplied and accumulated.
     */
    static const uint8_t taps[32] = {
        32, 31, 30, 29, 28, 27, 26, 25,
        24, 23, 22, 21, 20, 19, 18, 17,
        16, 15, 14, 13, 12, 11, 10, 9,
        8, 7, 6, 5, 4, 3, 2, 1 };

    /* This may result in some register spilling (and 4 unnecessary VMOVs). */
    const uint8x16_t t0 = vld1q_u8(taps);
    const uint8x16_t t1 = vld1q_u8(taps + 16);
    const uint8x8_t n_first_low = vget_low_u8(t0);
    const uint8x8_t n_first_high = vget_high_u8(t0);
    const uint8x8_t n_second_low = vget_low_u8(t1);
    const uint8x8_t n_second_high = vget_high_u8(t1);

    uint32x2_t adacc2, s2acc2, as;
    uint16x8_t adler, sum2;
    uint8x16_t d0, d1;

    uint32x4_t adacc = vdupq_n_u32(0);
    uint32x4_t s2acc = vdupq_n_u32(0);
```

```
    adacc = vsetq_lane_u32(s[0], adacc, 0);
    s2acc = vsetq_lane_u32(s[1], s2acc, 0);

    /*  Think of it as a vectorized form of the code implemented to
     * handle the tail (or a DO16 on steroids). But in this case
     * we handle 32 elements and better exploit the pipeline.
     */
    while (len >= 2) {
        d0 = vld1q_u8(buf);
        d1 = vld1q_u8(buf + 16);
        s2acc = vaddq_u32(s2acc, vshlq_n_u32(adacc, 5));
        adler = vpaddlq_u8(d0);
        adler = vpadalq_u8(adler, d1);
        sum2 = vmull_u8(n_first_low, vget_low_u8(d0));
        sum2 = vmlal_u8(sum2, n_first_high, vget_high_u8(d0));
        sum2 = vmlal_u8(sum2, n_second_low, vget_low_u8(d1));
        sum2 = vmlal_u8(sum2, n_second_high, vget_high_u8(d1));
        adacc = vpadalq_u16(adacc, adler);
        s2acc = vpadalq_u16(s2acc, sum2);
        len -= 2;
        buf += 32;
    }

    /* This is the same as before, but we only handle 16 elements as
     * we are almost done.
     */
    while (len > 0) {
        d0 = vld1q_u8(buf);
        s2acc = vaddq_u32(s2acc, vshlq_n_u32(adacc, 4));
        adler = vpaddlq_u8(d0);
        sum2 = vmull_u8(n_second_low, vget_low_u8(d0));
        sum2 = vmlal_u8(sum2, n_second_high, vget_high_u8(d0));
        adacc = vpadalq_u16(adacc, adler);
        s2acc = vpadalq_u16(s2acc, sum2);
        buf += 16;
        len--;
    }

    /* Combine the accumulated components (adler and sum2). */
    adacc2 = vpadd_u32(vget_low_u32(adacc), vget_high_u32(adacc));
    s2acc2 = vpadd_u32(vget_low_u32(s2acc), vget_high_u32(s2acc));
    as = vpadd_u32(adacc2, s2acc2);

    /* Store the results. */
    s[0] = vget_lane_u32(as, 0);
    s[1] = vget_lane_u32(as, 1);
}
```

The taps optimization that is referred to in the code comments works by computing the checksum of a vector of 32 elements where the n variable is known and fixed. This computed checksum is later recombined with another segment of 32 elements, rolling through the input data array. For more information, you can watch the BlinkOn 9: Optimizing image decoding on Arm presentation.

Elsewhere in the code, the expensive modulo operation is optimized so that it is only run when absolutely needed. The point at which the modulo is needed is just before the accumulated sum could possibly overflow the modulo value. This is calculated to be once every 5552 iterations.

The following table shows more information about the intrinsics in this example:

Table 3-2: Intrinsics used in the Adler-32 example

Intrinsic	Description
vaddq_u32	Vector add

Intrinsic	Description
vdupq_n_u32	Load all lanes of vector to the same literal value
vget_high_u32 vget_high_u8 vget_low_u32 vget_low_u8	Split vectors into two components
vget_lane_u32	Extract a single lane from a vector
vld1q_u8	Load a single vector or lane
vmlal_u8	Vector multiply and accumulate
vmull_u8	Vector multiply
vpadalq_u16 vpadalq_u8	Pairwise add and accumulate
vpadd_u32 vpaddlq_u8	Pairwise add
vsetq_lane_u32	Load a single lane of a vector from a literal
vshlq_n_u32	Vector shift left by constant

Results

Optimizing Adler-32 to use Neon intrinsics to perform SIMD arithmetic yielded significant performance improvements when this optimization started shipping in Chrome M63.

Tests in Armv8 showed an improvement of around 3x. For example, elapsed real time reduced from 350ms to 125ms for a 4096x4096 byte test executed 30 times.

This optimization alone yielded a performance boost for PNG decoding ranging from 5% to 18%.

Learn more

The following resources provide additional information about the Adler-32 optimization:

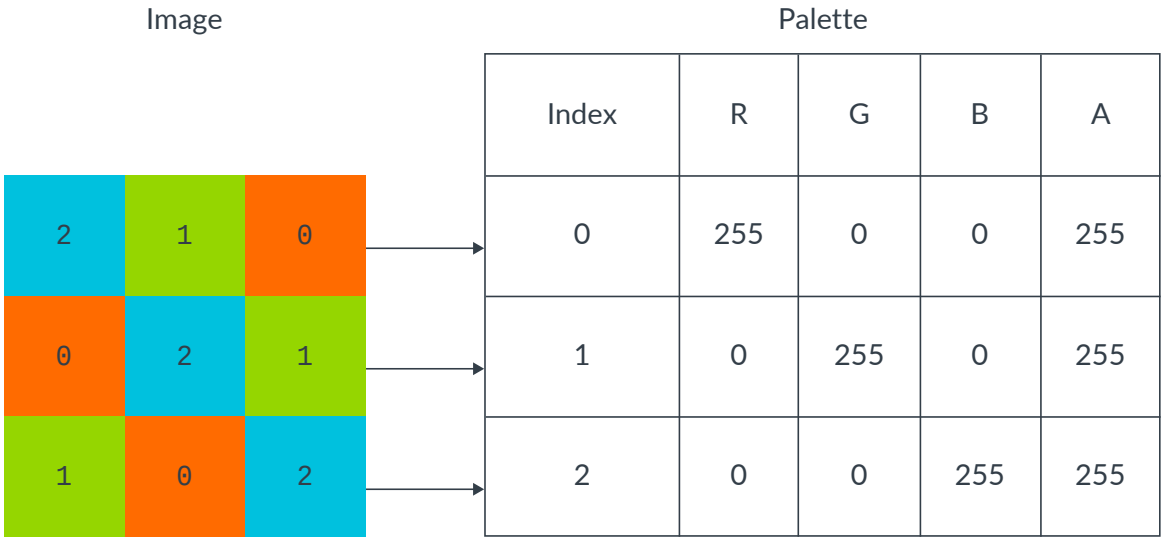
- Chromium Issue 688601: Optimize Adler-32 checksum
- Wikipedia: Adler-32
- BlinkOn 9: Optimizing image decoding on Arm

4. Chromium optimization: color palette expansion

In palettized PNG images, color information is not contained directly in the image's pixels. Instead, each pixel contains an index value into a palette of colors. This technique reduces the file size of PNG images, but means extra work must be done to display the PNG.

To render the PNG image, each palette index must be converted to an RGBA value by looking up that index in the palette. The following diagram shows how the palette maps different index values to RGB values.

Figure 4-1: Color palette expansion diagram



Unoptimized implementation

The original implementation of the palette expansion algorithm can be found in png_do_expand_palette(). The code iterates over every pixel, looking up each palette index (`*sp`) and adding the corresponding RGBA values to the output stream.

```
for (i = 0; i < row_width; i++)
{
    if ((int)(*sp) >= num_trans)
        *dp-- = 0xff;
    else
        *dp-- = trans_alpha[*sp];
    *dp-- = palette[*sp].blue;
    *dp-- = palette[*sp].green;
    *dp-- = palette[*sp].red;
    sp--;
}
```

Neon-optimized implementation

The optimized code uses Neon instructions to parallelize the data transfer and restructuring. The original code individually copied across each of the RGBA values from the index. The optimized code uses Neon intrinsics to construct a four-lane vector containing the R, G, B, and A values. This vector is then stored into memory. The optimized code using Neon intrinsics is as follows:

```
for(i = 0; i + 3 < row_width; i += 4) {
      uint32x4_t cur;
      png_bytep sp = *ssp - i, dp = *ddp - (i << 2);
      cur = vld1q_dup_u32 (riffled_palette + *(sp - 3));
      cur = vld1q_lane_u32(riffled_palette + *(sp - 2), cur, 1);
      cur = vld1q_lane_u32(riffled_palette + *(sp - 1), cur, 2);
      cur = vld1q_lane_u32(riffled_palette + *(sp), cur, 3);
      vst1q_u32((void *)dp, cur);
}
```

The following table shows more information about the intrinsics in this example:

Table 4-1: Intrinsics used in the color palette expansion example

Intrinsic	Description
vld1q_dup_u32	Load all lanes of a vector with the same value from memory
vld1q_lane_u32	Load a single lane of a vector with a value from memory
vst1q_u32	Store a vector into memory

Results

By using vectors to speed up the data transfer, performance gains in the range 10% to 30% have been observed.

This optimization started shipping in Chromium M66 and libpng version 1.6.36.

Learn more

The following resources provide additional information about the `png_do_expand_palette()` optimization:

- Chromium Issue 706134: Optimize png_do_expand_palette
- Wikipedia: Indexed color
- Portable Network Graphics (PNG) Specification (Second Edition)

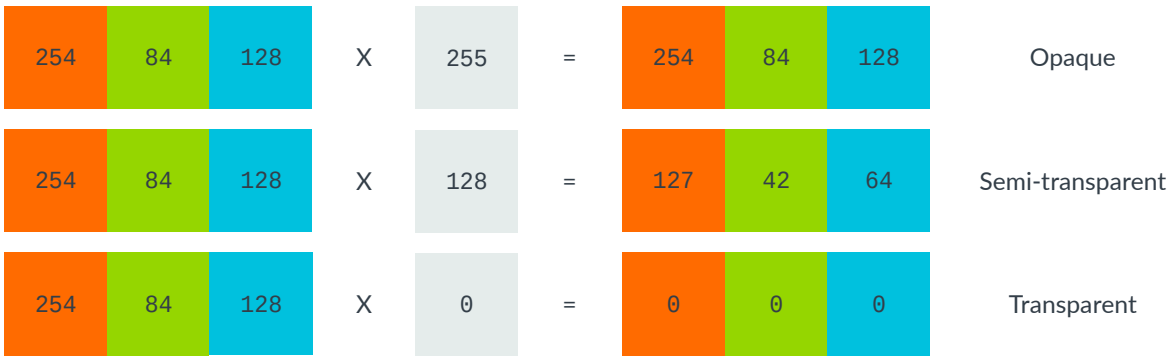
5. Chromium optimization: pre-multiplied alpha channel data

The color of each pixel in a PNG image is defined by an RGB triple. An additional value, called the alpha channel, specifies the opacity of the pixel. Each of the R, G, B, and A values are integers between 0 and 255. An alpha value of 0 means the pixel is transparent and does not appear in the final image. A value of 255 means the pixel is totally opaque and obscures any other image data in the same location.

When rendering a PNG image, the browser needs to calculate pre-multiplied alpha data. That is, the RGB data for each pixel must be multiplied by the corresponding alpha channel value. This calculation produces scaled RGB data that accounts for the opacity of the pixel.

The following diagram shows the same RGB pixel scaled by three different alpha values:

Figure 5-1: Pre-multiplied alpha channel data diagram



Each scaled color value is calculated as you can see in the following code:

```
Scaled_RGB_value = straight_rgb_value x (alpha_value / 255)
```

Unoptimized implementation

In Chromium, the code that performs this calculation is the `ImageFrame::setRGBAPremultiply()` function. Before Neon optimization, this function had the following implementation:

```
static inline void setRGBAPremultiply(PixelData* dest,
                                       unsigned r,
                                       unsigned g,
                                       unsigned b,
                                       unsigned a) {
    enum FractionControl { RoundFractionControl = 257 * 128 };

    if (a < 255) {
      unsigned alpha = a * 257;
      r = (r * alpha + RoundFractionControl) >> 16;
```

```
        g = (g * alpha + RoundFractionControl) >> 16;
        b = (b * alpha + RoundFractionControl) >> 16;
    }

    *dest = SkPackARGB32NoCheck(a, r, g, b);
}
```

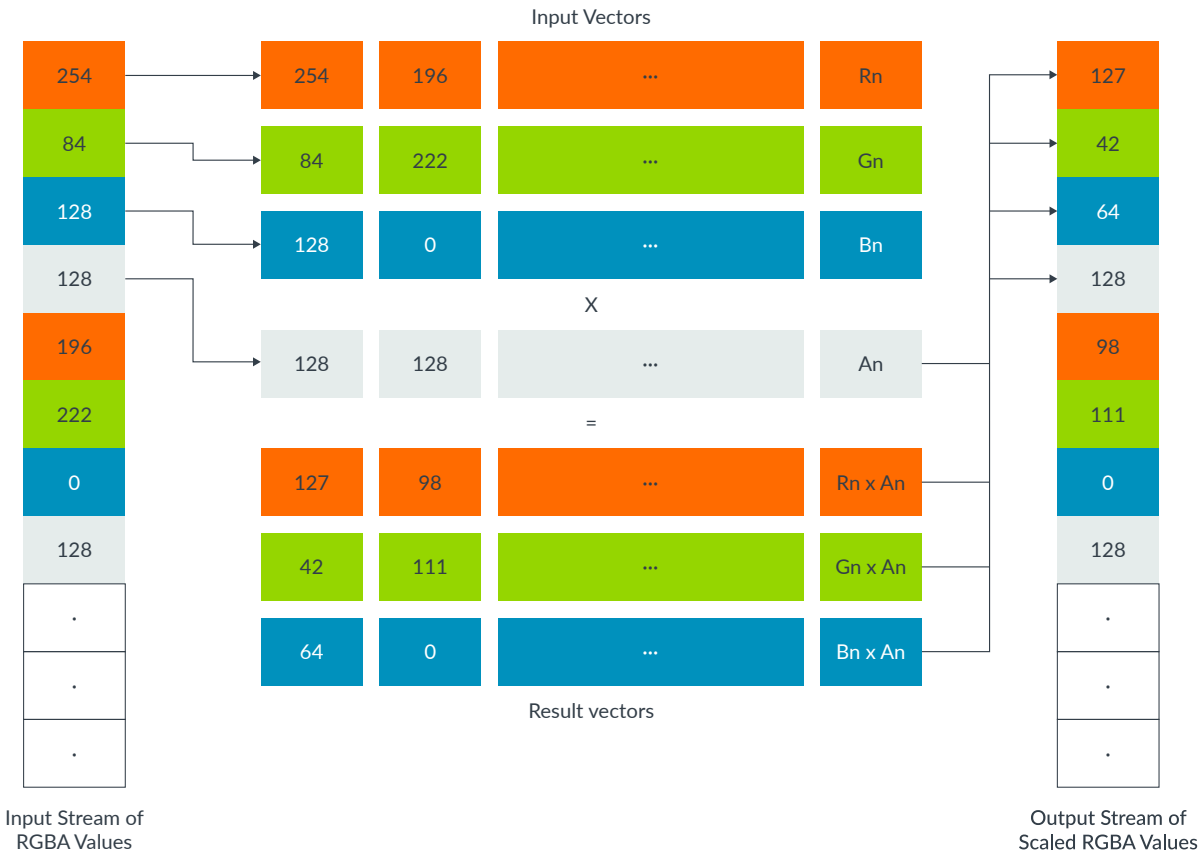
This unoptimized function operates on a single RGBA value at a time, multiplying each of the R, G, and B values by the alpha channel.

Neon-optimized implementation

The serial data processing performed in the unoptimized implementation provides an opportunity for Neon optimization. Rather than operating on a single data value at a time, we can:

- Load the RGBA data into separate R, G, B and A input vectors. Use a de-interleaved load. In this case, that means loading every fourth data value into the same register.
- Multiply each data lane with its corresponding alpha value simultaneously.
- Store the scaled data with an interleaved store. This means storing values from each of the four registers into adjacent memory locations, to produce an output stream of scaled RGBA data.

Figure 5-2: Input data for neon-optimized implementation diagram



The Neon optimized code is as follows:

```
static inline void SetRGBAPremultiplyRowNeon(png_bytep src_ptr,
                                             const int pixel_count,
                                             ImageFrame::PixelData* dst_pixel,
                                             unsigned* const alpha_mask) {
  // Input registers.
  uint8x8x4_t rgba;


  // Scale the color channel by alpha - the opacity coefficient.
  auto premultiply = [](uint8x8_t c, uint8x8_t a) {
    // First multiply the color by alpha, expanding to 16-bit (max 255*255).
    uint16x8_t ca = vmull_u8(c, a);
    // Now we need to round back down to 8-bit, returning (x+127)/255.
    // (x+127)/255 == (x + ((x+128)>>8) + 128)>>8.  This form is well suited
    // to NEON: vrshrq_n_u16(...,8) gives the inner (x+128)>>8, and
    // vraddhn_u16() both the outer add-shift and our conversion back to 8-bit.
    return vraddhn_u16(ca, vrshrq_n_u16(ca, 8));
  };

.
.
.

  // Main loop

  // Load data
  rgba = vld4_u8(src_ptr);

  // Premultiply with alpha channel
  rgba.val[0] = premultiply(rgba.val[0], rgba.val[3]);
  rgba.val[1] = premultiply(rgba.val[1], rgba.val[3]);
  rgba.val[2] = premultiply(rgba.val[2], rgba.val[3]);

  // Write back (interleaved) results to memory.
  vst4_u8(reinterpret_cast<uint8_t*>(dst_pixel), rgba);


}
```

The following table shows more information about the intrinsics in this example:

Table 5-1: Intrinsics used in the pre-multiplied alpha channel example

Intrinsic	Description
vmull_u8	Vector multiply
vraddhn_u16	Vector rounding addition
vrshrq_n_u16	Vector rounding shift right
vld4_u8	Load multiple 4-element structures to four vector registers
vst4_u8	Store multiple 4-element structures from four vector registers

Results

This optimization gave results in the region of 9% improvement.

Learn more

The following resources provide additional information about the `ImageFrame::setRGBAPremultiply()` optimization:

- Chromium Issue 702860: Optimize ImageFrame::setRGBAPremultiply
- The `SetRGBAPremultiplyRowNeon()` function in the Chromium codebase
- Wikipedia: Alpha compositing
- Arm Community Blog: Coding for Neon - Part 1: Load and Stores

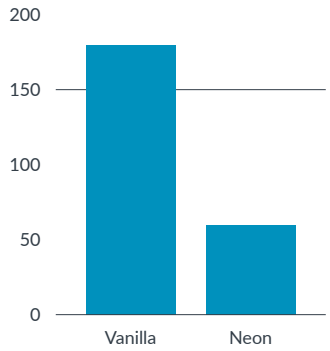
6. Chromium optimization: summary of results

This guide has shown how we identified optimization opportunities within the Chromium open-source codebase. The guide also provides detail about several specific optimizations made using Neon intrinsics.

One more notable optimization was a 20% increase in performance by optimizing `inflate_fast()` to use Neon intrinsics to perform long loads and stores in the byte array.

The result of all these optimizations was a 2.9x boost to PNG decoding performance. The following figure shows the decoding time improvement, in milliseconds, for test images comparing unoptimized zlib to Neon-optimized zlib:

Figure 6-1: PNG decoding performance diagram



Optimizations were validated using representative data sets. For PNG, we used three sets of test data:

- An internal data set for Chromium developers, with 92 images
- The public Kodak data set, with 24 images
- The public Google doodles data set, with 154 images

For more information about Neon programming in general, see the Neon Programmer's Guide for Armv8-A on the Arm Developer website.

For more information about Neon intrinsics, see the Neon Intrinsics Reference.

7. libTIFF optimization with Neon intrinsics

This section shows how Neon technology can improve performance in image processing applications. We use Neon intrinsics to optimize different aspects of the open-source Tag Image File Format (TIFF) image processing library, libTIFF.

Why libTIFF?

We chose to optimize libTIFF version 4.4.0 because it is an open-source project. This means that everyone has access to the source code.

Many large software projects use the libTIFF library, including the Android operating system and the Chrome web browser. This means that optimizations to libTIFF benefit a wide range of applications and users.

Finally, because libTIFF is an older library, there are many areas in the code that can be optimized for newer generations of hardware.

You can download the libTIFF 4.4.0 source code and read the build instructions.

Test platforms

Software performance, including Neon code, depends on many factors. These factors include the type of CPU on which the code runs, and also the operating system and its configuration settings.

To test the optimizations shown in this section, we used the following smartphones as target platforms:

- Samsung Galaxy S7, model SM-G930F released in 2016 with Android 7. This handset runs a Exynos 8890 octa-core chipset which includes Cortex-A53 cores running at up to 2.3 GHz.
- Google Pixel 4 XL, released in 2019 with Android 10. This handset runs a Snapdragon 855 octa-core chipset which includes Qualcomm Kryo 485 cores running at up to 2.8 GHz.

The Kernel Scheduler automatically moves application processes between slower or faster cores on Android, which affects performance. To ensure that the Kernel Scheduler did not affect measurements, on the Galaxy S7 we enabled only one of the faster cores and set the **Kernel Scaling Governor** to **Performance** mode. This setting forces the CPU frequency to the maximum possible. Tests on the Pixel 4 XL were performed without this change.


Note

If you change the **Kernel Scaling Governor** configuration setting you might need to build the Android operating system from source code. This is outside the scope of this guide.

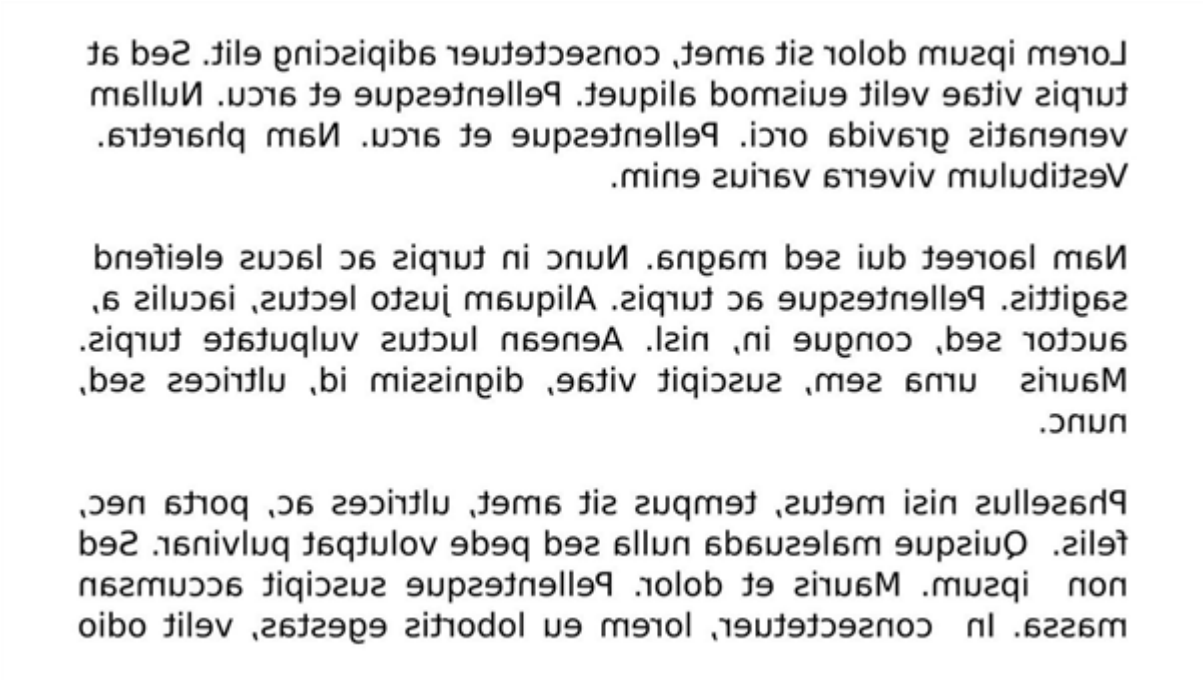
Performance tests

To test the performance of libTIFF, we developed a custom Android app using two different images. Each image exercises different areas of code within libTIFF.

To measure performance, we first process the images using the original non-Neon version of libTIFF, and then using the Neon-optimized version. When processing is complete, the application reports performance statistics for both libTIFF versions.

The first image is a horizontally-flipped grayscale image with black and white text. The image uses 8 bits per pixel (8BPP). The orientation of this image is top-right, so the first pixel in data is the top-right of the image. This image allows us to use two Neon optimizations: one for converting 8BPP image data to 32BPP, and one for the horizontal flip.

Figure 7-1: Test image 1: Horizontally-flipped grayscale image



The second image is similar to the first image, but with varying background colors behind the text. This image is stored in CMYK format, and lets us test the Neon optimization for converting CMYK to RGBA.

Figure 7-2: Test image 2: CMYK color image

The tests use the default libTIFF compiler build options, which specify the `-O2` optimization flag. The build command line is too complicated to include in full. The simplified version for one of the source files is as follows:

```
$ aarch64-linux-android24-clang -O2 -c SOURCE_FILE.c -fPIC -o SOURCE_FILE.o
```

8. libTIFF optimization: grayscale to RGBA conversion

The first test image is black and white with data stored as 8BPP grayscale. When the `TIFFImageRGBARead` function extracts this image, it converts the pixels to RGBA.

Unoptimized implementation

The following code shows the original libTIFF implementation:

```
static void putgreytile(TIFFRGBAImage* img, uint32_t* cp,
                        uint32_t x, uint32_t y, uint32_t w, uint32_t h,
                        int32_t fromskew, int32_t toskew, unsigned char* pp)
{
  int samplesperpixel = img->samplesperpixel;
  uint32_t** BWmap = img->BWmap;
  // 1. For all lines of the image..
  for( ; h > 0; --h) {
    // 2. For all pixels across a line..
    for (x = w; x > 0; --x) {
      // 3. Convert the 8-bit pixel value into a 32-bit RGBA pixel (1 pixel at a
 time)
      *cp++ = BWmap[*pp][0];
      pp += samplesperpixel;
    }
    cp += toskew;
    pp += fromskew;
  }
}
```

The `TIFFImageRGBARead` function loads the image, expanding each 8-bit pixel to 32 bits, 8 bits per channel.

The `BWmap` variable is a lookup table that converts 8-bit grayscale values to 32-bit values. With the sample image, it takes an 8-bit value B and returns a 32-bit value with B copied to each 8-bit channel and the value 255 inserted into the alpha channel.

Neon-optimized implementation

The Neon-optimized implementation processes the data for four pixels simultaneously. For simplicity, this implementation optimizes the most frequent use case where the values `toskew` and `fromskew` are zero.

```
// 1. Does the image match our requirements for optimization and are we to use Neon?
//    (tif_packbitsmode is a tag that is set by the app).
//    Check to see if there are a multiple of 4 pixels. This is to ensure that we
//    can process four pixels in one iteration.
//    Each pixel is 32 bits, so four pixels fit exactly into the 128-bit Neon
 registers.
uint32_t n = h * w;
if (img->tif->tif_packbitsmode == PackBits_Neon
    && toskew == 0
    && fromskew == 0
    && (n & 0x3) == 0)
{
  n >>= 2;
  uint32x4_t p = vdupq_n_u32(0);
```

```
  while(n-- > 0) {

    // 2. Use the BWmap lookup table to convert the pixel to 32-bit value.
    //    Each 32-bit pixel is loaded into one of 4 lanes of the 128-bit register.
    p = vld1q_lane_u32(BWmap[*pp], p, 0);
    pp += samplesperpixel;
    p = vld1q_lane_u32(BWmap[*pp], p, 1);
    pp += samplesperpixel;
    p = vld1q_lane_u32(BWmap[*pp], p, 2);
    pp += samplesperpixel;
    p = vld1q_lane_u32(BWmap[*pp], p, 3);
    pp += samplesperpixel;

    // 3. Write out four pixels with a single 128-bit write operation.
    vst1q_u32(cp, p);
    cp += 4;
  }
}
```

The following table shows more information about the intrinsics in this example:

Table: Intrinsics used in the grayscale to RGBA conversion example

Intrinsic	Description
vdupq_n_u32	Duplicate vector element
vld1q_lane_u32	Load multiple single-element structures
vst1q_u32	Store multiple single-element structures to memory

Results

The performance results for using Neon to optimize grayscale to RGBA conversion are as follows:

Table: Performance results

Test platform	Original code runtime (ms)	Neon optimized code runtime (ms)	Performance improvement
Galaxy S7	6.09	3.16	1.92x
Pixel 4 XL	4.27	2.53	1.68x

Performance times are for 1000 iterations in all cases.

9. libTIFF optimization: horizontal image flip

TIFF allows images to be stored in any orthogonal layout. The first pixel in memory can be the top-left, top-right, bottom-left, or bottom-right of the image. For the first test image, the TIFF image is right-to-left and we need the pixels in a left-to-right layout. We need to perform a horizontal flip of the image.

Unoptimized implementation

The following code shows the original libTIFF implementation of the image flip operation:

```
uint32_t line;
// 1. For an image of width w and height h, for all lines in the image, do the
 following.
for (line = 0; line < h; line++) {
  uint32_t *left = raster + (line * w);
  uint32_t *right = left + w - 1;
  // 2. Swap the pixel at the beginning of the line with the pixel at the end,
  //    then work inwards, swapping as we go.
  while ( left < right ) {
    // 3. Swap two pixels on the same line
    uint32_t temp = *left;
    *left = *right;
    *right = temp;
    left++;
    right--;
  }
}
```

This code contains two small loops that reverse the ordering of the pixels line-by-line.

Each pixel is 32-bits, in RGBA color format, so the lines of pixels are reversed 32-bits at a time rather than byte-by-byte.

Neon-optimized implementation

The Neon-optimized implementation reverses the pixels in a row of the image by creating an index table. Each iteration of the loop swaps four pixels from the left of the row with four pixels from the right.

For simplicity, Neon optimization is only applied if the image is a multiple of 8 pixels wide.

```
// For all lines of the image..
for (line = 0; line < h; line++) {
    uint32_t *left = raster + (line * w);
    uint32_t *right = left + w;
    right -= 4;
    // Create an index table to obtain pixel information four pixels, based
    // on an offset from the left and right base addresses.
```

```
    //
    // This index table is used later by the table lookup intrinsic vqtbl1q_u8.
    //
    // The indices swap the pixel order preserving channel ordering
    // within the pixels, remembering that each pixel is 4 bytes.
    //
    // For example, if we have four RGBA pixels W, X, Y and Z, we swap
    // them to Z, Y, X and W.
    //
    // This example uses decimal values to aid comprehension.
    //
    // uint8_t reverseIndices[16] = {
    //   Fetch pixel Z's RGBA components (indices R=12, G=13, B=14 and A=15)
    //      and place them in indices 0 to 3:
    //   [ 0] 0x0C = 12, // Z.Red   = 12th byte in the input
    //   [ 1] 0x0D = 13, // Z.Green = 13th byte in the input
    //   [ 2] 0x0E = 14, // Z.Blue  = 14th byte in the input
    //   [ 3] 0x0F = 15, // Z.Alpha = 15th byte in the input
    //   Fetch pixel Y's RGBA components:
    //   [ 4] 0x08 =  8,
    //   [ 5] 0x09 =  9,
    //   [ 6] 0x0A = 10,
    //   [ 7] 0x0B = 11,
    //   Fetch pixel X's RGBA components:
    //   [ 8] 0x04 =  4,
    //   [ 9] 0x05 =  5,
    //   [10] 0x06 =  6,
    //   [11] 0x07 =  7,
    //   Fetch pixel W's RGBA components:
    //   [12] 0x00 =  0,
    //   [13] 0x01 =  1,
    //   [14] 0x02 =  2,
    //   [15] 0x03 =  3 };
    uint8x8_t reverse1 = vcreate_u8(0x0B0A09080F0E0D0Cull);
    uint8x8_t reverse2 = vcreate_u8(0x0302010007060504ull);
    uint8x16_t reverseIndices = vcombine_u8(reverse1, reverse2);
    // Each loop iteration swaps four pixels from the left with
    // four pixels from the right, reversing the order within each
    // batch of four pixels.
    while ( left < right ) {
        // Load pixels from the left and reverse their order
        uint8x16_t leftPixels = vld1q_u8((uint8_t*)left);
        uint8x16_t reversedLeftPixels = vqtbl1q_u8(leftPixels, reverseIndices);
        // Load pixels from the right and reverse their order
        uint8x16_t rightPixels = vld1q_u8((uint8_t*)right);
        uint8x16_t reversedRightPixels = vqtbl1q_u8(rightPixels, reverseIndices);
        // Copy the right-hand pixels to the left and the left-hand pixels
        // to the right
        vst1q_u8((uint8_t*)left, reversedRightPixels);
        vst1q_u8((uint8_t*)right, reversedLeftPixels);
        left += 4;
        right -= 4;
    }
}
```

The following table shows more information about the intrinsics in this example:

Table: Intrinsics used in the image flip example

Intrinsic	Description
vcreate_u8	Create a vector from a literal value
vcombine_u8	Join two smaller vectors into a single larger vector
vqtbl1q_u8	Table vector lookup
vld1q_u8	Load multiple single-element structures

Intrinsic	Description
vst1q_u8	Store multiple single-element structures to memory

Results

The following table shows performance results for using Neon to optimize the horizontal flip operation:

Table: Performance results

Test platform	Original code runtime (ms)	Neon optimized code runtime (ms)	Performance improvement
Galaxy S7	3.12	2.71	1.15x
Pixel 4 XL	8.25	6.28	1.31x

Performance times are for 1000 iterations in all cases.

10. libTIFF optimization: CMYK to RGBA conversion

The second test image stores pixels in CMYK (Cyan, Magenta, Yellow and Black) format. CMYK is used in printing. Popular image editing programs often support loading and saving files as CMYK.

TIFF supports automatic conversion of CMYK to RGBA using the `TIFFReadRGBA` interface.

To convert CMYK to RGB, libtiff uses the following calculations:

- `R = 255 x (1 - C) x (1 - K)`
- `G = 255 x (1 - M) x (1 - K)`
- `B = 255 x (1 - Y) x (1 - K)`

The alpha channel A is always output as 255.

Unoptimized implementation

The following code shows the original libTIFF implementation of the CMYK to RGBA conversion:

```
// The original code uses a macro UNROLL8 to unroll code and
// each iteration processes eight pixels.
//
// The macro actually contains a for loop that iterates over a single line
// (the w parameter is the width of the image).

#define UNROLL8(w, op1, op2) { \
  uint32_t _x; \
// For the whole width of the image..
  for (_x = w; _x >= 8; _x -= 8) { \
  op1; \
// Repeat for 8 pixels..
  REPEAT8(op2); \
  } \
// For any pixels left over..
  if (_x > 0) { \
    op1; \
    CASE8(_x,op2); \
  } \
} // end of macro UNROLL8()

// For each line of the image (h is the image height)
for( ; h > 0; --h) {
  // Convert 8 pixels from CMYK to RGBA (A is always 255)
  UNROLL8(w, NOP, {
    k = 255 - pp[3];
    r = (k*(255-pp[0]))/255;
    g = (k*(255-pp[1]))/255;
    b = (k*(255-pp[2]))/255;
    // Write each pixel to memory one at a time
    *cp++ = PACK(r, g, b);
    pp += samplesperpixel
  });
  cp += toskew;
  pp += fromskew;
}
```

Neon-optimized implementation

The Neon-optimized implementation uses intrinsics to perform calculations for multiple pixels simultaneously.

As with the other libTIFF optimizations in this guide, the full version of the code uses an `if` statement to check whether to use the original code or the Neon code. For example, we only optimize the case where skewing is not required, that is when the values `toskew` and `fromskew` are zero. For clarity, the `if` statement is not shown here.

```
// Loop over all pixels of the image
uint32_t np = w * h;
uint32_t* endp = cp + np;

// Indices for VTBL that duplicate each pixels K value
uint8x8_t dupK1 = vcreate_u8(0xff06ff06ff06ff06ull);
uint8x8_t dupK2 = vcreate_u8(0xff0eff0eff0eff0eull);
uint8x16_t kindices = vcombine_u8(dupK1, dupK2);

// Indices to obtain the final results
uint8_t resultIndices[16] = {0,1,2,-1,4,5,6,-1,8,9,10,-1,12,13,14,-1};
while(cp < endp) {
    // 16 copies of 255
    uint8x16_t v255 = vdupq_n_u8 (255);
    // load 4 pixels (each pixel is 4 bytes with the CMYK values)
    uint8x16_t src_u8 = vld1q_u8(pp);
    // perform (255 - x) on each component
    // each vsubl is working on 2 pixels
    uint16x8_t subl = vsubl_u8(vget_low_u8(v255), vget_low_u8(src_u8));
    uint16x8_t subh = vsubl_high_u8(v255, src_u8);
    // duplicate k element from each pixel in subl
    uint8x16_t kl = vqtbl1q_u8(subl, kindices);
    uint8x16_t kh = vqtbl1q_u8(subh, kindices);
    // multiply (255 - x) by (255 - k)
    uint16x8_t ml = vmulq_u16(kl, subl);
    uint16x8_t mh = vmulq_u16(kh, subh);
    // the results we need are in the low 8 bits of the uint16 elements
    // combine results and result (throwing away all the upper halves of all the
 uint16)
    uint8x16_t idx = vld1q_u8 (resultIndices);
    uint16x8_t resultl = ml / 255;
    uint16x8_t resulth = mh / 255;
    uint8x16_t packed = vuzp1q_u8 (vreinterpretq_u8_u16 (resultl),
                                   vreinterpretq_u8_u16 (resulth));
    // wherever the index is -1, we take the value from v255 (we return 255 in
 alpha)
    uint8x16_t pixels = vqtbx1q_u8 (v255, packed, idx);
    // store the four RGBA pixels and advance the pointers/counters
    vst1q_u8((uint8_t*)cp, pixels);
    cp += 4;
    pp += samplesperpixel * 4;
}
```

The following table shows more information about the intrinsics in this example:

Table: Intrinsics used in the CMYK to RGBA example

Intrinsic	Description
vcombine_u8	Join two smaller vectors into a single larger vector

Intrinsic	Description
vcreate_u8	Create a vector from a literal value
vdupq_n_u8	Load all lanes of vector to the same literal value
vget_low_u8	Split vectors into two components
vld1q_u8	Load multiple single-element structures
vmulq_u16	Vector multiply
vqtbl1q_u8	Table vector lookup
vqtbx1q_u8	Table vector lookup extension
vreinterpretq_u8_u16	Vector reinterpret cast operation
vst1q_u8	Store multiple single-element structures to memory
vsubl_high_u8	Vector subtract using upper half of vector elements
vsubl_u8	Vector subtract
vuzp1q_u8	Unzip vectors, reading corresponding even-numbered vector elements from two source vectors

Results

The performance results for using Neon to optimize the CMYK to RGBA conversion are as follows:

The following table shows performance results for using Neon to optimize the CMYK to RGBA conversion:

Table: Performance results

Test platform	Original code runtime (ms)	Neon optimized code runtime (ms)	Performance improvement
Galaxy S7	13.31	7.70	1.72x
Pixel 4 XL	11.80	5.89	2.00x

Performance times are for 1000 iterations in all cases.

11. libTIFF optimization: auto-vectorization and compiler options

The compiler options used to build a library can have a large effect on the performance of your program.

In particular, compiler options can enable or disable auto-vectorization features in your compiler that automatically optimize your code to take advantage of Neon.

To demonstrate the effect of auto-vectorization, we built and benchmarked the original non-Neon version of libTIFF with two different compiler optimization options:

-O0
: Minimum optimization, auto-vectorization disabled.

-O2
: High optimization, auto-vectorization enabled.

The following table shows the results of running the original non-Neon version of libTIFF built with different compiler options. The tests are the same benchmark tests used earlier in this guide, running on the Galaxy S7.

Table: Performance results for different compiler options

Optimization	Runtime with -O0 (ms)	Runtime with -O2 (ms)	Performance improvement
Grayscale to RGBA	17.97	6.09	3.5x
Flip	10.84	3.12	3.4x
CMYK to RGBA	46.95	13.31	3.5x