Neon Intrinsics Case Study: Optimizing PNG Performance in Chromium
Neon Intrinsics: Optimizing PNG Performance in Chromium

Copyright © 2019 Arm Limited (or its affiliates). All rights reserved.


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

http://www.arm.com
# Contents

1 Overview ................................................................................................................................................................................................................... 5  
  1.1. What are Neon intrinsics? ........................................................................................................................................................................................................ 5  
  1.2. Why Chromium? ........................................................................................................................................................................................................... 5  
  1.3. Why PNG? .......................................................................................................................................................................................................... 6  
  1.4. Introducing Bobby the budgie ........................................................................................................................................................................... 6  

2 Adler-32 ..................................................................................................................................................................................................................... 9  
  2.1. Unoptimized implementation ............................................................................................................................................................................. 10  
  2.2. Neon-optimized implementation ................................................................................................................................................................. 10  
  2.3. Results .................................................................................................................................................................................................................. 12  
  2.4. Further information ....................................................................................................................................................................................... 12  

3 Color palette expansion ..................................................................................................................................................................................... 13  
  3.1. Unoptimized implementation ............................................................................................................................................................................. 13  
  3.2. Neon-optimized implementation ................................................................................................................................................................. 14  
  3.3. Results .................................................................................................................................................................................................................. 14  
  3.4. Further information ....................................................................................................................................................................................... 14  

4 Premultiplied alpha channel data ................................................................................................................................................................... 15  
  4.1. Unoptimized implementation ............................................................................................................................................................................. 15  
  4.2. Neon-optimized implementation ................................................................................................................................................................. 16  
  4.3. Results .................................................................................................................................................................................................................. 18  
  4.4. Further information ....................................................................................................................................................................................... 18  

5 Summary ................................................................................................................................................................................................................. 19
1 Overview

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

The goal of this guide is to demonstrate to programmers who might be unfamiliar with Neon how they can use intrinsics in their code to enable SIMD (Single Instruction, Multiple Data) processing. Using Neon in this way can bring huge performance benefits, as we will discover in this case study.

1.1. 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, for example, that you can have a single source code implementation that can be built for both 32-bit and 64-bit targets.

1.2. Why Chromium?

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

Chromium provides the basis for Google Chrome, the world’s most popular web browser in terms of user numbers. Any performance improvements we were able to make to the Chromium codebase had the potential to 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, such as programming with Neon intrinsics, it often helps to have examples to learn from. We hope that the examples provided in this guide will prove especially helpful because they can be seen in the context of a complete, real-world, codebase.
1.3. Why PNG?

The next question we asked was: where should we look in the Chromium code to make optimizations? With over 25 million lines of code, we needed to pick a specific area to target. When looking at the type of workloads web browser 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 53% of mobile users abandon sites that take over 3 seconds to load, so optimizing image load times (and therefore page load times) should bring tangible benefits.

PNG was developed as an improved, non-patented replacement for Graphics Interchange Format (GIF) and is the standard for transparent images in the web. It is also a popular format for web graphics in general. This led to Arm's decision to investigate opportunities for Neon optimization in PNG image processing.

1.4. Introducing Bobby the budgie

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

This image of a budgerigar has complex textures, a reasonably large size, and a transparent background, which makes it a good test case for investigating optimizations to the PNG decoding process.

![Image of a budgerigar](Image source: [Penubag](https://commons.wikimedia.org/wiki/File:Penubag.png) [Public domain], via Wikimedia Commons)

The first thing to note is that all PNG images are not created equal. There are a number of 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 around 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 will reduce file size, but also potentially degrade 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 may degrade image quality if the original image contains more colors than the maximum allowed by the palette. 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>
<thead>
<tr>
<th>Image</th>
<th>File size</th>
<th>Number of colors</th>
<th>Palette or TrueColor?</th>
<th>Filters?</th>
<th>Compression</th>
<th>Encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original_Bobby.PNG</td>
<td>2.7M</td>
<td>211787</td>
<td>TrueColor</td>
<td>Yes</td>
<td>zlib</td>
<td>libpng</td>
</tr>
<tr>
<td>Palette_Bobby.PNG</td>
<td>0.9M</td>
<td>256</td>
<td>Palette</td>
<td>No</td>
<td>zlib</td>
<td>libpng</td>
</tr>
<tr>
<td>Zopfli_Bobby.PNG</td>
<td>2.6M</td>
<td>211787</td>
<td>TrueColor</td>
<td>Yes</td>
<td>Zopfli</td>
<td>ZopfliPNG</td>
</tr>
</tbody>
</table>

To obtain performance data for each of these three images, we used the Linux `perf` tool to profile `ContentShell`.

**Original_Bobby.PNG**

- Image has pre-compression filters (2.7MB)

<table>
<thead>
<tr>
<th>Lib</th>
<th>Command</th>
<th>SharedObj</th>
<th>Method</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>inflate_fast...................</td>
<td>1.96</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>adler32.......................</td>
<td>0.88</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>liblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.45</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>liblink</td>
<td>png_read_filter_row_up........</td>
<td>0.03*</td>
</tr>
</tbody>
</table>

**Palette_Bobby.PNG**

- Image has no pre-compression filters (0.9MB)

<table>
<thead>
<tr>
<th>Lib</th>
<th>Command</th>
<th>SharedObj</th>
<th>Method</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>libpng</td>
<td>TileWorker</td>
<td>liblink</td>
<td>cr_png_do_expand_palette.....</td>
<td>0.88</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>inflate_fast...................</td>
<td>0.62</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>liblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.49</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>adler32.......................</td>
<td>0.31</td>
</tr>
</tbody>
</table>

**Zopfli_Bobby.PNG**

- Image was optimized using zopfli (2.6MB)

<table>
<thead>
<tr>
<th>Lib</th>
<th>Command</th>
<th>SharedObj</th>
<th>Method</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>inflate_fast...................</td>
<td>3.06</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>liblink</td>
<td>adler32.......................</td>
<td>1.36</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>liblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.70</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>liblink</td>
<td>png_read_filter_row_up........</td>
<td>0.48*</td>
</tr>
</tbody>
</table>

This data helped us identify the zlib library as a good target for our optimization efforts, as it contains a number of methods that contribute significantly to performance.
In addition, zlib was 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 to name but a few. This meant that any performance improvements we could achieve in zlib would yield performance improvements for a large number of users.

- Released in 1995, the zlib library has a relatively old codebase. Older codebases with areas that might not have been modified in many years, are likely to provide more opportunities for improvement.

- The zlib library did not contain any existing optimizations for Arm, which meant there were likely to be a wide range of improvements that could be made.
2 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.

In the PNG format, Adler-32 is used for the uncompressed PNG data while 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.

The Adler-32 checksum function can therefore be expressed as follows:

\[
A = 1 + D_1 + D_2 + \ldots + D_n \pmod{65521}
\]

\[
B = (1 + D_1) + (1 + D_1 + D_2) + \ldots + (1 + D_1 + D_2 + \ldots + D_n) \pmod{65521}
\]

\[
= nD_1 + (n-1)D_2 + (n-2)D_3 + \ldots + D_n + n \pmod{65521}
\]

\[
\text{Adler-32}(D) = (B \times 65536) + A
\]

For example, to calculate the Adler-32 checksum of the ASCII string Neon:

<table>
<thead>
<tr>
<th>Character</th>
<th>Decimal ASCII code</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>78</td>
<td>((1 + 78) % 65521 = 79)</td>
<td>((0 + 79) % 65521 = 79)</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>((79 + 101) % 65521 = 180)</td>
<td>((79 + 180) % 65521 = 259)</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>((180 + 111) % 65521 = 291)</td>
<td>((259 + 291) % 65521 = 550)</td>
</tr>
<tr>
<td>n</td>
<td>110</td>
<td>((291 + 110) % 65521 = 401)</td>
<td>((550 + 401) % 65521 = 951)</td>
</tr>
</tbody>
</table>

The decimal Adler-32 checksum is calculated as follows:

\[
\text{Adler-32} = (B \times 65536) + A
\]

\[
= (951 \times 65536) + 401
\]

\[
= 62,324,736 + 401
\]

\[
= 62,325,137
\]

Or, in hexadecimal:

\[
\text{Adler-32} = (B \times 00010000) + A
\]

\[
= (03B7 \times 00010000) + 0191
\]

\[
= 03B70000 + 0191
\]

\[
= 03B70191
\]
2.1. Unoptimized implementation

Wikipedia provides the following simplistic implementation of the Adler-32 algorithm:

```c
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.

2.2. Neon-optimized implementation

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

```c
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;
```
```c
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 = vpaddfq_u8(d0);
adler = vpadalq_u8(adler, d1);
    sum2 = vmullq_u8(n_first_low, vget_low_u8(d0));
    sum2 = vmulq_u8(sum2, n_second_low, vget_low_u8(d1));
    sum2 = vmulq_u8(sum2, n_second_hi, vget_hi_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 = vpaddfq_u8(d0);
    sum2 = vmullq_u8(n_second_low, vget_low_u8(d0));
    sum2 = vmulq_u8(sum2, n_second_hi, vget_hi_u8(d0));
adacc = vpadalq_u16(adacc, adler);
s2acc = vpadalq_u16(s2acc, sum2);
    buf += 16;
    len--;
}

/* Combine the accumulated components (adler and sum2). */
adacc2 = vpaddd_u32(vget_low_u32(adacc), vget_hi_u32(adacc));
s2acc2 = vpaddd_u32(vget_low_u32(s2acc), vget_hi_u32(s2acc));
as = vpaddd_u32(adacc2, s2acc2);

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

The “taps” optimization 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, then later on recombining the computed checksum 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. This is at the point just before the accumulated sum could overflow the modulo value, which is calculated to be once every 5552 iterations.
Additional information about the intrinsics used:

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaddq_u32</td>
<td>Vector add.</td>
</tr>
<tr>
<td>vdupq_n_u32</td>
<td>Load all lanes of vector to the same literal value.</td>
</tr>
<tr>
<td>vget__hi gh_u32</td>
<td>Split vectors into two components.</td>
</tr>
<tr>
<td>vget__hi gh_u8</td>
<td></td>
</tr>
<tr>
<td>vget__lw ow_u32</td>
<td></td>
</tr>
<tr>
<td>vget__lw ow_u8</td>
<td></td>
</tr>
<tr>
<td>vget_l ane_u32</td>
<td>Extract a single lane from a vector.</td>
</tr>
<tr>
<td>vl d1q_u8</td>
<td>Load a single vector or lane.</td>
</tr>
<tr>
<td>vm al __u8</td>
<td>Vector multiply and accumulate.</td>
</tr>
<tr>
<td>vmul l __u8</td>
<td>Vector multiply.</td>
</tr>
<tr>
<td>vpadal q_u16</td>
<td>Pairwise add and accumulate.</td>
</tr>
<tr>
<td>vpadal q_u8</td>
<td></td>
</tr>
<tr>
<td>vpadd_u32</td>
<td>Pairwise add.</td>
</tr>
<tr>
<td>vpaddfl q_u8</td>
<td></td>
</tr>
<tr>
<td>vset q_l ane_u32</td>
<td>Load a single lane of a vector from a literal.</td>
</tr>
<tr>
<td>vshl q_n_u32</td>
<td>Vector shift left by constant.</td>
</tr>
</tbody>
</table>

### 2.3. Results

Optimizing Adler-32 to use Neon intrinsics to perform SIMD arithmetic yielded significant performance improvements when it 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%.

### 2.4. Further information

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
3 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.

<table>
<thead>
<tr>
<th>Image</th>
<th>Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

3.1. 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.

```c
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--;
}```
3.2. Neon-optimized implementation

The optimized code uses Neon instructions to parallelize the data transfer and restructuring. Rather than individually copy across the each of the RGBA values from the index, this optimized code uses Neon intrinsics to construct a 4-lane vector containing the R, G, B and A values. This vector is then stored into memory.

```c
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);
}
```

Additional information about the intrinsics used:

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vl d1q_dup_u32</td>
<td>Load all lanes of a vector with the same value from memory.</td>
</tr>
<tr>
<td>vl d1q_lane_u32</td>
<td>Load a single lane of a vector with a value from memory.</td>
</tr>
<tr>
<td>vst1q_u32</td>
<td>Store a vector into memory.</td>
</tr>
</tbody>
</table>

3.3. 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.

3.4. Further information

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

- Chromium Issue 706134: Optimize png_do_expand_palette
- Wikipedia: Indexed color
4 Premultiplied 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 represented by a value 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 premultiplied alpha data. That is, the RGB data for each pixel must be multiplied by the corresponding alpha channel value to produce 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.

Each scaled color value is calculated as follows:

\[
\text{Scaled_RGB_value} = \text{straight_rgb_value} \times (\text{alpha_value} / 255)
\]

4.1. Unoptimized implementation

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

```cpp
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.

### 4.2. Neon-optimized implementation

This type of serial data processing 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, using a de-interleaved load (in this case, 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 (storing values from each of the four registers into adjacent memory locations) to produce an output stream of scaled RGBA data.

![Diagram](image.png)
The Neon optimized code is as follows:

```c
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]);

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

Additional information about the intrinsics used:

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmull_u8</td>
<td>Vector multiply.</td>
</tr>
<tr>
<td>vraddhn_u16</td>
<td>Vector rounding addition.</td>
</tr>
<tr>
<td>vrshrq_n_u16</td>
<td>Vector rounding shift right.</td>
</tr>
<tr>
<td>vld4_u8</td>
<td>Load multiple 4-element structures to four vector registers.</td>
</tr>
<tr>
<td>vst4_u8</td>
<td>Store multiple 4-element structures from four vector registers.</td>
</tr>
</tbody>
</table>
4.3. Results

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

4.4. Further information

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
5 Summary

This guide has shown how we identified optimization opportunities within the Chromium open source codebase. It also provides detail about a number of specific optimizations made using Neon intrinsics.

One additional 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 end result of all these optimizations was a 2.9x boost to PNG decoding performance. The following figure shows the decoding time improvement (in ms) for test images comparing vanilla (unoptimized) zlib to Neon-optimized zlib:

![Decoding time improvement graph]

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.