# Why zlib?

<table>
<thead>
<tr>
<th>Zlib</th>
<th>Context</th>
<th>Problem statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used everywhere (libpng, Skia, freetype, cronet, Firefox, Chrome, linux kernel, android, iOS, JDK, git, etc). Old code base released in 1995. Written in K&amp;R C style.</td>
<td>Lacks any optimizations for ARM CPUs.</td>
<td>Identify potential optimization candidates and verify positive effects in Chromium.</td>
</tr>
</tbody>
</table>
Previous art

- Cloudflare
- Intel
- Zlib-ng
Before deepening the fork...

- Performed some benchmarking.
- Contacted each project.
- Mixed results (1 project never replied back).
Before forking...

- Performed some benchmarking.
- Contacted each project.
- Mixed results (1 project never replied back).

None focused on **decompression*** or had ARM specific optimizations.

*Important for a Web Browser.
Meet Mr. Parrot

PNGs rely on zlib
● Transparent.
● Pre-filters.
● High-res.

Source: https://upload.wikimedia.org/wikipedia/commons/3/3f/ZebraHighRes.png
Parrots are not created equal

Original: 2.7MB
Palette: 0.8MB
Zopfli: 2.6MB
## Perf to the rescue

### 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>libblink</td>
<td>inflate_fast</td>
<td>1.96</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>libblink</td>
<td>adler32</td>
<td>0.88</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>libblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.45</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>libblink</td>
<td>png_read_filter_row_up</td>
<td>0.03*</td>
</tr>
</tbody>
</table>

### 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>libblink</td>
<td>inflate_fast</td>
<td>3.06</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>libblink</td>
<td>adler32</td>
<td>1.36</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>libblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.70</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>libblink</td>
<td>png_read_filter_row_up</td>
<td>0.48*</td>
</tr>
</tbody>
</table>

### 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>libblink</td>
<td>cr_png_do_expand_palette</td>
<td>0.88</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>libblink</td>
<td>inflate_fast</td>
<td>0.62</td>
</tr>
<tr>
<td>blink</td>
<td>TileWorker</td>
<td>libblink</td>
<td>ImageFrame::setRGBAPremultiply</td>
<td>0.49</td>
</tr>
<tr>
<td>zlib</td>
<td>TileWorker</td>
<td>libblink</td>
<td>adler32</td>
<td>0.31</td>
</tr>
</tbody>
</table>
NEON: Advanced SIMD
(Single Instruction Multiple Data)
● Optional on ARMv7.
● Mandatory on ARMv8.
Registers

ARMv7
- 16 registers@128 bits: Q0 - Q15.
- 32 registers@64bits: D0 - D31.
- Varied set of instructions: load, store, add, mul, etc.

ARMv8
- 32 registers@128 bits: Q0 - Q31.
- 32 registers@64bits: D0 - D31.
- 32 registers@32bits: S0 - S31.
- 32 registers@8bits: H0 - H31.
- Varied set of instructions: load, store, add, mul, etc.
An example: VADD.I16 Q0, Q1, Q2
Entropy & Compression
Entertaining definition

https://www.youtube.com/watch?v=l49MHwooAVQ
Formal definition

Shannon Entropy

\[ H = - \sum_i p_i \log_b p_i \]

Where:
\( p_i \): probability of character \( i \) appearing in the stream of characters.

Practical explanation

a) HTML

b) JPEG
Practical visualization

./binwalk -E file

a) HTML: 0.68

b) JPEG: 0.95
Decompression optimizations
Adler-32 checksum

\[ A = 1 + D_1 + D_2 + \ldots + D_n \pmod{65521} \]
\[ B = (1 + D_1) + (1 + D_1 + D_2) + \ldots + (1 + D_1 + D_2 + \ldots + D_n) \pmod{65521} \]
\[ = n \times D_1 + (n-1) \times D_2 + (n-2) \times D_3 + \ldots + D_n + n \pmod{65521} \]

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

https://en.wikipedia.org/wiki/Adler-32
Adler-32 simplistic implementation

```c
// From: https://en.wikipedia.org/wiki/Adler-32
const int MOD_ADLER = 65521;
unsigned long naive_adler32(unsigned char *data,
                          unsigned long len)
{
    uint32_t a = 1, b = 0;
    unsigned long index;

    for (index = 0; index < len; ++index) {
        a = (a + data[index]) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }

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

https://en.wikipedia.org/wiki/Adler-32
Adler-32: problems

- Zlib’s Adler-32 was more than 7x faster than naive implementation.
- It is hard to vectorize the following computation:

```c
void accum(uint32_t *pair, const unsigned char *buf, unsigned int len)
{
    unsigned int i;
    for (i = 0; i < len; ++i) {
        pair[0] += buf[i];
        pair[1] += pair[0];
    }
}
```
Adler-32: technical drawing (Jan 2017)
Adler-32

‘Taps’ to the rescue

Assembly:
https://godbolt.org/g/KMeBAJ
Adler-32: Intel got some love too!

Add SSSE3 implementation of the adler32 checksum, suitable for both large workloads, and small workloads commonly seen during PNG image decoding. Add a NEON implementation.

Speed is comparable to the serial adler32 computation but near 64 bytes of input data, the SIMD code paths begin to be faster than the serial path: 3x faster at 256 bytes of input data, to ~8x faster for 1M of input data (~4x on ARMv8 NEON).

For the PNG 140 image corpus, PNG decoding speed is ~8% faster on average on the desktop machines tested, and ~2% on an ARMv8 Pixel C Android (N) tablet, https://crbug.com/762564#c1

Update x86.(c,h) to runtime detect SSSE3 support and use it to enable the adler32_simd code path and update inflate.c to call x86_check_features(). Update the name mangler file names.h for the new symbols added, add FIXME about simd.patch.

Ignore data alignment in the SSSE3 case since unaligned access is no longer penalized on current generation Intel CPU. Use it in the NEON case however to avoid the extra costs of unaligned memory access on ARMv8/v7.

NEON credits: the v_s1/s2 vector component accumulate code was provided by Adenilson Cavalcanti. The uint16 column vector sum code is from libdeflate with corrections to process NMAX input bytes which improves performance by 3% for large buffers.

https://bugs.chromium.org/p/chromium/issues/detail?id=688601
fast_chunk

- Second candidate in the perf profiling was **inflate_fast**.
- Very **high level** idea: perform long loads/stores in the byte array.
- Average **20% faster**!
- Shipping on M62.
- Original patch by Simon Hosie.

https://bugs.chromium.org/p/chromium/issues/detail?id=697280
CRC-32

- YMMV on PNGs (from 1 to 5%).
- Remember it is used while decompressing web content (29% boost for gzipped content).
- ARMv8-a has a crc32 instruction (from 3 to 10x faster than zlib’s crc32 C code).
- Shipping on M66.

https://bugs.chromium.org/p/chromium/issues/detail?id=709716
Results: Chromium’s zlib*

* c-zlib
Arm: zlib format 1.4x
Arm: gzip format 1.5x
Arm: c-zlib X Vanilla

ARM: decompression vs compression speed

- Compression
- Decompression

Speed boost

File types
x86: c-zlib X Vanilla
We were missing compression...
Bonus: Compression on Arm
Slide-hash: NEON

- Using NEON instruction vqsubq.
- Works on 8x 16bits chunks.
- Perf gain of 5%.

https://chromium-review.googlesource.com/1136940
insert-string: crypto CRC-32

- Using ARMv8-a instruction crc32.
- Works on 1x 32bits chunks.
- Perf gain of 24%.

https://chromium-review.googlesource.com/c/chromium/src/+/1173262
Arm: current state

- Compression: average 1.36x faster, but 1.4x faster for HTML.
- Decompression: average 1.6x faster (gzip), but 1.8x faster for HTML.
Conclusions
Conclusions

- There is plenty of life left even in an old code base.
- NEON optimizations can yield a *huge* impact.
- It pays up to work in a lower layer.
- OSS love: Intel got it too.
Chromium’s zlib: c-zlib

- Decompression: 1.7x to 2x faster.
- Compression: 1.3x to 1.4x faster.
- Both ARM & x86 are supported.
- Highly tested (i.e. cronet, fuzzers).
- Widely deployed (over 1 billion users).
- Open to performance & security patches.
Chromium’s zlib: c-zlib

- Decompression: 1.7x to 2x faster.
- Compression: 1.3x to 1.4x faster.
- Both ARM & x86 are supported.
- Highly tested (i.e. cronet, fuzzers).
- Widely deployed (over 1 billion users).
- Open to performance & security patches.

Zlib users should consider moving to Chromium’s zlib.
Resources

a) Slides: https://goo.gl/vaZA9o
b) Performance benchmarks: https://goo.gl/qLVdvh
c) Code: https://cs.chromium.org/chromium/src/third_party/zlib/
“This is how the open-source model works: building upon the work of others is far more efficient than rewriting everything.”

Jean-loup Gailly (zlib author)

https://slashdot.org/story/00/03/10/1043247/jean-loup-gailly-on-gzip-go-and-mandrake