FJXL and FPNGE

Very Fast SIMD lossless image encoders

Luca Versari (veluca@google.com)

Overview

In November 2021, QOI took the Internet by storm.

Main selling point: simpler and faster than PNG.

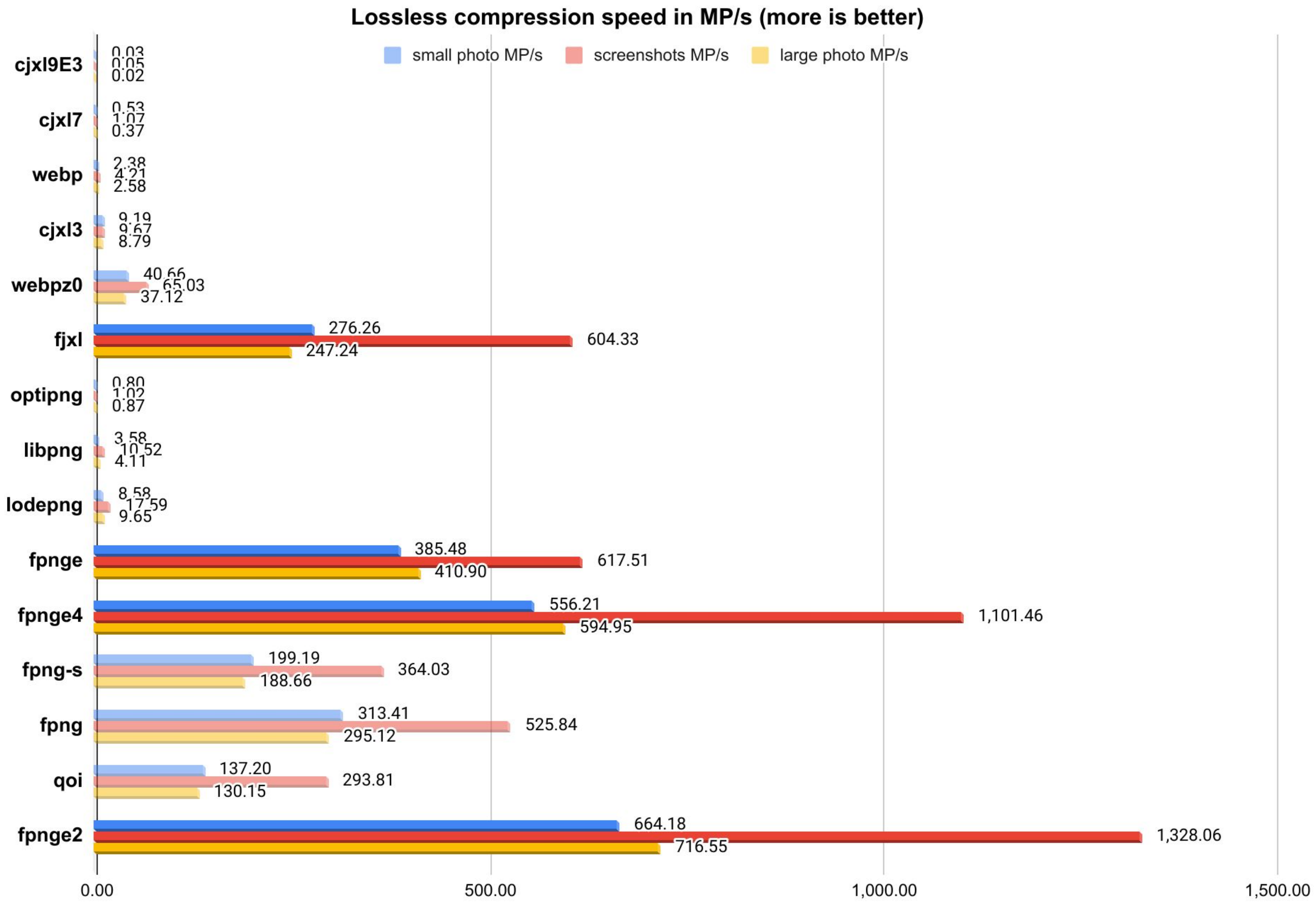
I decided to find out whether you can do better with existing formats.

The result of this effort are FJXL and FPNGE: new fast encoders 100% conformant to the JPEG XL and PNG specification, respectively.

Results

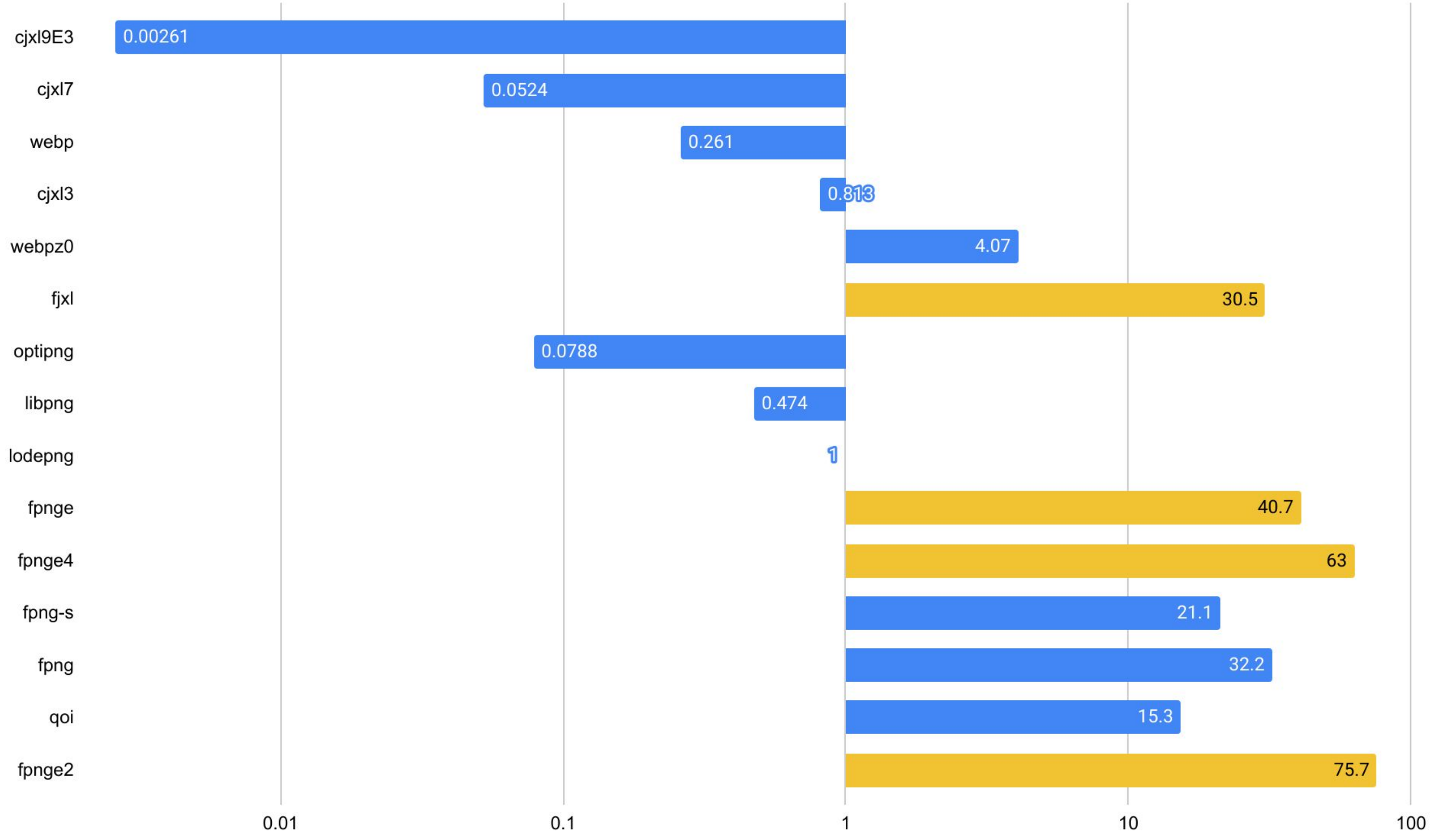
small photos: https://goo.gl/cmHIkl

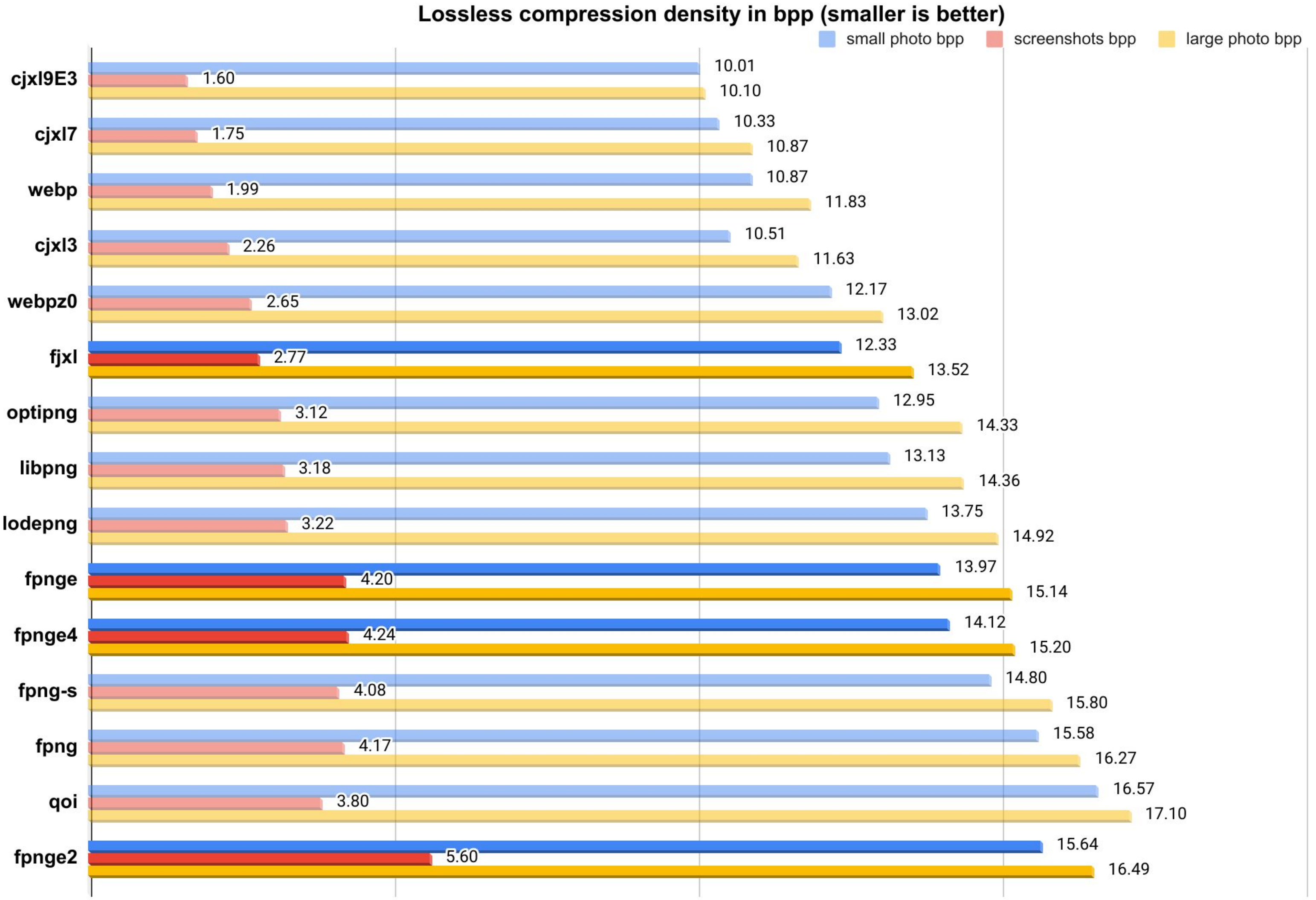
large photos: JPEG XL CTC images, excluding non-photographic images



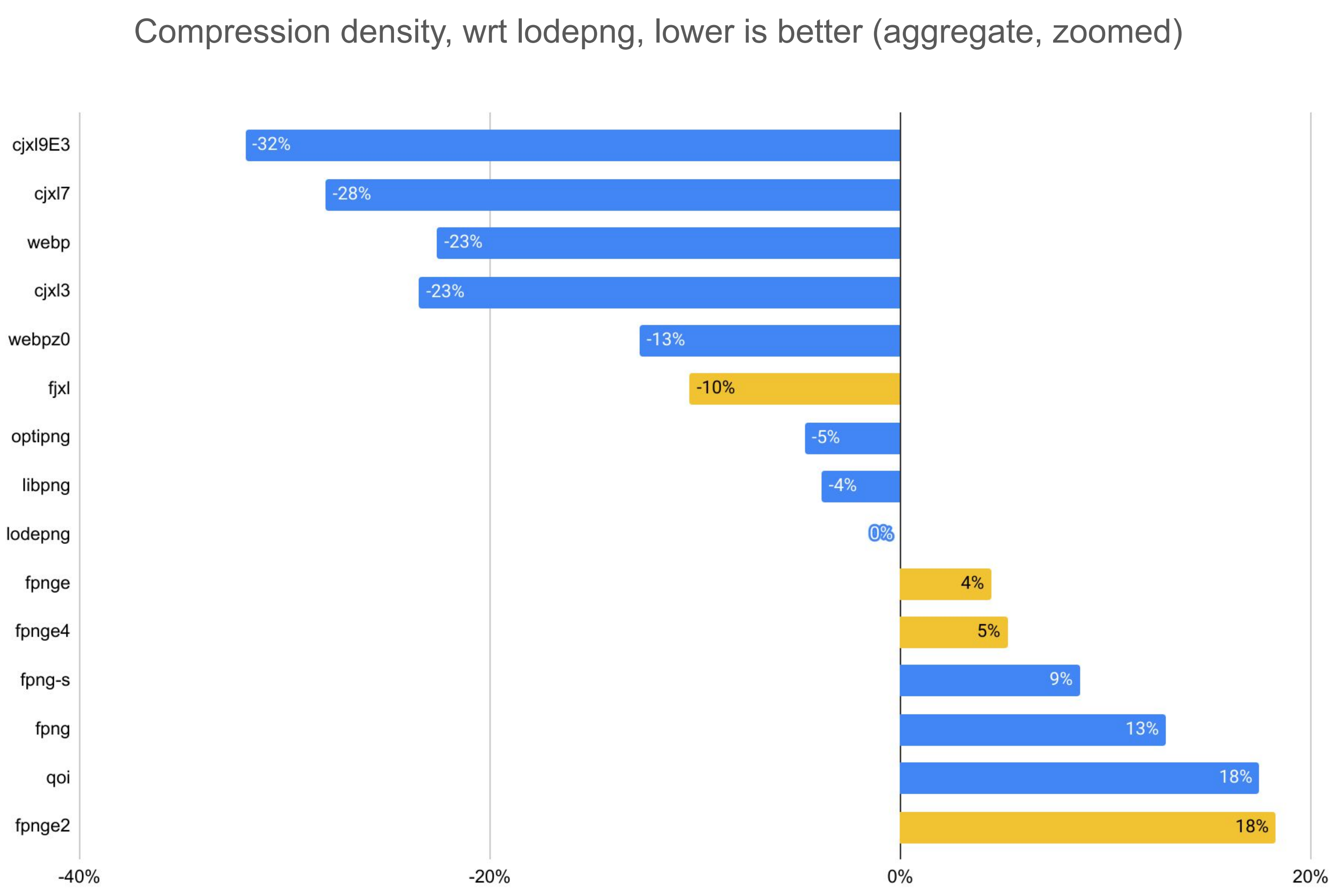
AMD Ryzen 7 5800X 8-Core Processor, 1 thread

Compression speed, wrt lodepng, higher is better (log scale)





Lossless compression density in bpp (smaller is better)
small photo bpp
screenshots bpp
large photo bpp
cjxl9E3 10.01 1.60 10.10
cjxl7 10.33 1.75 10.87
webp 10.87 1.99 11.83
cjxl3 10.51 2.26 11.63
webpz0 12.17 2.65 13.02
fjxl 12.33 2.77 13.52
optipng 12.95 3.12 14.33
libpng 13.13 3.18 14.36
lodepng 13.75 3.22 14.92
fpnge 13.97 4.20 15.14
fpnge4 14.12 4.24 15.20
fpng-s 14.80 4.08 15.80
fpng 15.58 4.17 16.27
qoi 16.57 3.80 17.10
fpnge2 15.64 5.60 16.49



Compression density, wrt lodepng, lower is better (aggregate, zoomed)
cjxl9E3
cjxl7
webp
cjxl3
webpz0
fjxl
optipng
libpng
lodepng
fpnge
fpnge4
fpng-s
fpng
qoi
fpnge2
-32%
-28%
-23%
-23%
-13%
-10%
-5%
-4%
0%
4%
5%
9%
13%
18%
18%
-40%
-20%
0%
20%

Density vs speed, compared to lodepng (aggregate)
relative speed
relative density
100
10
1
0.1
0.01
10%
0%
-10%
-20%
-30%
fpnge2
fpnge4
fpnge
fpng
fpng-s
qoi
fjxl
webpz0
lodepng
libpng
cjxl3
webp
optipng
cjxl7
cjxl9E3

FJXL

FJXL - JPEG XL lossless subset overview

- (optional) Palettization is applied
- Image is divided into 256x256 tiles
- Color transform (YCoCg) applied (8 -> 9 bit range)
- Channels are separated into planes
- Prediction is applied (ClampedGradient) (9 -> 10 bit range)
- LZ77 can be applied (in this encoder, just run-length encoding)
- “Hybrid UInt encoding” is applied on raw symbols, split into symbol + bits
- Prefix coding is applied on symbols

Input is 8-bit, but the added bits require switching to 16-bit arithmetic. On AVX2, that’s 16 integers per vector.

FJXL - palette detection

- Hash table with 65k possible entries
- Any collision -> no palette
- Palette is sorted by luma to make the prediction still work well
- At most 256 palette entries are used

On non-palette-friendly images, this fails quickly (birthday paradox says after ~256 distinct pixels).

On palette images, encoding 1 channel rather than 4 more than compensates the cost of detection.

FJXL - prediction

TL	T
L	

ClampedGradient predictor:

- predicts a pixel with an estimate of the gradient: T + L - TL
- … but without extrapolation, i.e. clamped to the [min(T, L, TL); max(T, L, TL)] range

On the encoder side, simple to (auto)vectorize.

FJXL - RLE SIMD-fication

Very simple SIMD approach to run length encoding:

- if the current vector of values is identical to the last value from the previous vector, increment run count and skip producing output
- if not, emit a LZ77 copy length + distance symbol (if run length > 0) and encode the current vector raw

FJXL - Hybrid UInt Encoding

Given a number (written in base 2)

$$1b_{p-1}b_{p-2}\ldots b_1b_0$$

we split it into a symbol that represents p (or a special symbol that represents 0) + p raw bits "$b_{p-1}b_{p-2}\ldots b_1b_0$"

This requires a fast-log2 operation; __builtin_clz() does the job, can be emulated for AVX2 vectors with some table lookups in a 16-entry table (vpshufb)

FJXL - Prefix coding

Usual prefix coding. Optimization: sample a few parts of the image (at random) to produce an image-adapted table.

We only have <16 symbols. We can SIMDfy prefix coding by doing table lookups with vpshufb.

We still need to concatenate all the Huffman and raw bits into a single bit stream. This could be done with 32 calls to a PutBits function, but SIMDfication provides a significant speed up: can be done with a sequence of bitwise operations to reduce to just 4 PutBits.

FPNGE

FPNGE - PNG vs JXL differences

- No division in tiles
- No color transforms
- One of 5 predictors (“filters”) is chosen per row
- Channels are interleaved, not split into planes
    - Makes RLE somewhat less effective
    - Requires masking to disable unused channels, or specialized code paths
- All operations are byte-wise, with wraparound
    - Can use 8-bit integers, i.e. 32 integers per vector
- No Hybrid Uint Encoding - Huffman raw alphabet has 256 symbols
- Two different checksums of image data (Adler32 and CRC32)

FPNGE - PNG filters

- Filter 0: noop
- Filter 1: subtract a
- Filter 2: subtract b
- Filter 3: subtract (a+b)/2
- Filter 4: subtract the Paeth predictor

c	b
a	

```
p = a+b-c;
pa = |p - a|;
pb = |p - b|;
pc = |p - c|;
if pa <= pb && pa <= pc PAETH = a;
else if pb <= pc PAETH = b;
else PAETH = c;
```

FPNGE - Paeth SIMDfication

Intermediate quantities for Paeth do **not** use wraparound: problematic for SIMD.

Alternative, equivalent formulation with just 8-bit intermediate quantities:

```
bc = b - c;
ca = c - a;
pa = c < b ? bc : -bc;
pb = a < c ? ca : -ca;
pc = (a < c) == (c < b) ? (bc >= ca ? bc - ca : ca - bc) : 255;
PAETH = pa <= pb && pa <= pc ? a : pb <= pc ? b : c;
```

FPNGE - Filter choice

- libpng: filter that minimizes the sum of absolute values of residuals for the row
- fpnge: filter that minimizes the bit cost of the row (doing full mock-encodes)
- fpnge4: always use the Paeth predictor
- fpnge2: always use the Top predictor

FPNGE - Huffman coding

Fast Huffman coding in FJXL works because we have at most 16 raw symbols, which fit in a single vector pair.

For PNG, we can pick a custom Huffman table so that:

- Symbols [0, 16) (0000xxxx in binary) have their own Huffman code
- Symbols [240, 256) (1111xxxx in binary) have their own Huffman code
- All other symbols (yyyyxxxx in binary) have LUT[yyyy]xxxx as their Huffman code, i.e. lowest 4 bits are copied as-is.

The Huffman table can then fit in 3 vector pairs, which still allows fast lookups with vpshufb + vpblendvb.

Code

Code

FJXL: https://github.com/libjxl/libjxl/tree/main/experimental/fast_lossless

FPNGE: https://github.com/veluca93/fpnge

Both are single-file encoders (+ main file).

FJXL has AVX2, ARM NEON and plain-C++ implementations.

FPNGE is AVX2 only.

Questions?