GFWX: GOOD, FAST WAVELET CODEC
ICT TECH REPORT ICT-TR-01-2016

Graham Fyffe

University of Southern California Institute for Creative Technologies

ABSTRACT
Wavelet image compression is a popular paradigm for lossy and lossless image coding, and the wavelet transform, quantization, and entropy encoding steps are well studied. Efficient implementation is straightforward for the first two steps using e.g. lifting and uniform scalar deadzone quantization, but entropy encoding is typically carried out using complex context modeling and arithmetic coding. We propose a simple entropy encoding scheme for wavelet coefficients based on limited-length Golomb-Rice codes, and we propose two simple context schemes for selecting the Golomb parameter; one slightly faster than the other. We also propose a simple solution to the rounding problem in integer wavelet transforms, allowing the use of integer transforms for both lossless and lossy compression instead of resorting to floating point. If the input is Bayer patterned data, we include an additional lifting step allowing lossy compression without demosaicing. We demonstrate that a straightforward implementation of a complete codec in under 1000 lines of C++ is several times faster than JPEG 2000 while producing similar file sizes, without sacrificing certain desirable features such as downsampled decoding and progressive decoding of incomplete data streams. We provide an implementation in under 1000 lines of C++ source code using only standard C++ libraries, with many of the features that make JPEG 2000 attractive including optional downsampled decoding, and progressive decoding of incomplete data streams. Sections 2 through 5 describe the steps of our proposed pipeline, and Section 6 discusses the implementation and evaluation.

2. INTEGER COLOR TRANSFORM
Our pipeline begins with an optional user-programmable color transform. Let the input data represent one or more non-interleaved image layers, each pixel having one or more interleaved channels. The memory layout of the data is hence ordered by layers, then rows, then columns, and finally interleaved channels. Conceptually, each spatial image location has one data value per layer per interleaved channel, which we collectively call channels. We support any transform that can be expressed as a sequence of steps of the following form:

\[ C_i \leftarrow C_i + \left[ \frac{1}{s} \sum_{t=1}^{n} k_t C_{j_t} \right]_{0}^{\downarrow} \]

where \( C_i \) is the value of channel \( i \), \( s \) is an integer divisor, \( j_t \neq i \) are summand channel indices, \( k_t \) are integer weights, and \( \left[ x \right]_{0}^{\downarrow} = \text{sign}(x) \left| x \right| \) represents rounding towards zero. This general form is trivially reversible, and allows many popular color transforms to be implemented. For example, the YUV transform can be implemented for RGB input as:

\[
\begin{align*}
C_0 &\leftarrow C_0 + \left[ -C_1 \right]_{0}^{\downarrow}; \\
C_2 &\leftarrow C_2 + \left[ -C_1 \right]_{0}^{\downarrow}; \\
C_1 &\leftarrow C_1 + \left[ \frac{1}{2} (C_0 + C_2) \right]_{0}^{\downarrow},
\end{align*}
\]

and the \( A_{7,10} \) transform can be implemented as:

\[
\begin{align*}
C_0 &\leftarrow C_0 + \left[ -C_1 \right]_{0}^{\downarrow}; \\
C_2 &\leftarrow C_2 + \left[ \frac{1}{2} (-C_0 - 2C_1) \right]_{0}^{\downarrow}; \\
C_1 &\leftarrow C_1 + \left[ \frac{1}{8} (3C_0 + 2C_2) \right]_{0}^{\downarrow}.
\end{align*}
\]

Index Terms— Wavelets, image compression, JPEG 2000, Golomb-Rice codes, Bayer pattern compression.
During encoding, the result of the color transform is stored in a temporary buffer with a larger integer precision than the input data, which facilitates later steps that produce intermediate values having additional bits of dynamic range. For example, 8-bit data is transformed into a 16-bit temporary buffer, and 16-bit data is transformed into a 32-bit temporary buffer.

3. INTEGER WAVELET TRANSFORM

After the color transform, we apply an in-place lifting scheme to compute the wavelet transform per channel. Like JPEG 2000, we recommend using the 5/3 wavelet for lossless encoding, and the 9/7 wavelet for lossy encoding, however we support both modes for both encoding types. Normally, the 9/7 wavelet transform is implemented for lossy encoding using floating-point arithmetic, because integer wavelet transforms based on lifting suffer from a rounding problem, where rounding of residuals prevents proper smoothing of low frequency values. We solve this using what amounts to fixed point arithmetic, without any impact on encoding or decoding speed. We simply multiply the input values by 8, which we fold into the color transform step. The extra 3 bits of precision is enough to smooth the low frequency values. Using a value greater than 8 is also possible, but may cause integer overflow in our implementation of the 9/7 wavelet transform. After the transform, we divide the output by 8, which we fold into the quantization step. We employ this scheme for both 5/3 and 9/7 wavelets, but only for lossy compression, as it is not strictly reversible. We additionally clamp the cubic predictions of the 9/7 wavelet transform to lie within the range defined by the two center samples of the cubic, which reduces ringing artifacts and slightly improves compression. If the input is Bayer patterned data, it is known that the first level of a wavelet transform effectively decorrelates the color channels into the LL, LH, HL, and HH subbands, however the LH, HL, and HH subbands still exhibit significant spatial correlation. Therefore in the case of Bayer data we repeat the entire wavelet transform on the first-level LH, HL and HH subbands, improving decorrelation and hence compression.

4. QUANTIZATION

After the wavelet transform, lossy compression may be achieved by quantizing the coefficients. We employ uniform scalar deadzone quantization (USDQ) to encode wavelet coefficients in a block-channel using limited-length Golomb-Rice codes as described in Subsections 5.1 and 5.2. After encoding, we tightly pack the encoded block-channels in sequence, exploiting the word alignment of the encoded buffers for efficiency. For speed and simplicity, we encode whole coefficients instead of multiple bitplane passes, which means we lose the embedded encoding property prevalent in most wavelet image codecs. While embedded encoders are progressive in quality (where each transmitted bit optimally improves a quality measure such as PSNR), our proposed encoder is progressive in resolution (where each transmitted level doubles the resolution of the decodable image). We argue that this is a desirable feature for today’s so-called responsive media applications, where a variety of devices may request different resolutions from a single image resource.

5. ENTROPY ENCODING

After quantization, we encode the wavelet coefficients in level order, starting from the DC coefficient. Within each level, we divide the pixels into blocks that are 2\textsuperscript{b} pixels on each side for some user-selected \(b\). We encode blocks in scanline order, and we encode the channels within a block sequentially and independently. This allows the encoding task to be divided into a number of parallel encoding subtasks equal to the number of blocks times the number of channels, called block-channels. Since the length of the encoded block-channels is unknown beforehand, we divide the remaining encoding buffer into equal parts on word boundaries and encode each block-channel into its own part. For each block-channel, we encode wavelet coefficients in scanline order using limited-length Golomb-Rice codes as described in Subsections 5.1 and 5.2.

5.1. Limited-Length Golomb-Rice Codes

We encode the wavelet coefficients in a block-channel using limited-length Golomb-Rice codes, or zero run codes when the probability of zero is high. Golomb-Rice coding with zero runs is popular for encoding prediction residuals in e.g. JPEG-LS, and we find them equally effective for encoding wavelet coefficients. Rather than constructing a custom code table based on symbol statistics as in Huffman coding, Golomb coding implements a computationally simple code with just one free integer parameter, called the Golomb parameter. An integer \(x \geq 0\) is encoded as the unary representation of \([x/m]\) (so many zeros followed by a one) followed by the binary representation of \(x \mod m\), where \(m\) is the Golomb parameter. Golomb-Rice codes are Golomb codes where \(m\) is a power of two, which admit extremely sim-
The second is to encode the index of an additional bit. This limiting scheme has several deficiencies: it requires advance knowledge of the range of $x$; it is redundant as any $x < lm$ has two representations; and the implied probability distribution changes abruptly from geometric to flat at the limiting threshold. We propose instead a more gradual limiting scheme. Simply, if $|x/m| > l$, then we emit $l$ zeros to signify an escape, and then emit $x$ in binary using $32$ additional bits. This limiting scheme has several deficiencies: it requires advance knowledge of the range of $x$; it is redundant as any $x < lm$ has two representations; and the implied probability distribution changes abruptly from geometric to flat at the limiting threshold. We propose instead a more gradual limiting scheme. Simply, if $|x/m| > l$, then we emit $l$ zeros to signify an escape, and then we recursively encode $x - lm$ using a larger modulus (we found $16m$ works well). The advantages of this technique are that the range of $x$ need not be known; every $x$ has exactly one representation; and the implied probability distribution flattens gradually as $x$ increases, which improves compression in our tests. For signed data, such as wavelet coefficients, two schemes are common. The first is to interleave positive and negative values in sequence, i.e. 0, 1, −1, 2, −2, . . . and encode the index of $x$ in this sequence, which we call interleaved codes. The second is to encode $|x|$ followed by a sign bit if $x \neq 0$, which we call signed codes. We note that the implied probability distribution differs between these two schemes, and so rather than choosing one over the other, we use both. We encode zero run lengths using Golomb-Rice codes, and we encode wavelet coefficients using either interleaved Golomb-Rice codes or signed Golomb-Rice codes. Switching between these three methods and selecting the Golomb parameter in each case is described in the next subsection.

5.2. Selecting the Golomb Parameter

Optimal selection of the Golomb parameter $m$ is possible if the data comes from a geometrically distributed source with known mean $\mu$. A popular selection strategy that is always close to optimal is to let $m = 2^k$ where $2^k \leq \mu < 2^{k+1}$. However, the distribution of wavelet coefficients in real-world images is not strictly geometric, and therefore we propose a scheme based on the first moment $\mu_1 = \mathbb{E}[|x|]$ and second moment $\mu_2 = \mathbb{E}[|x|^2]$, to capture richer distribution characteristics. We propose two methods to estimate $(\mu_1, \mu_2)$. The first method is to examine a neighborhood around a coefficient shown in Fig. 1 and compute the following estimates:

\[
\hat{\mu}_1 = \text{round}\left(\frac{16}{\sum_{i \in N} w_i} \sum_{i \in N} w_i |y_i|\right);
\]

\[
\hat{\mu}_2 = \text{round}\left(\frac{16}{\sum_{i \in N} w_i} \sum_{i \in N} w_i \min(4096, |y_i|^2)\right),
\]

where $N$ is the set of neighbors that exist considering image boundaries and scanline order precedence, $y_i$ is the coefficient at the $i$th neighbor, and $w_i$ is its weight. Scaling the moments by 16 lets us use integer arithmetic, and limiting the values to 4096 avoids overflow later on. The second method, which is somewhat faster but less effective, is to update a running estimate of the two moments after each value $x$ is encoded:

\[
\hat{\mu}_1 \leftarrow \text{round}\left(\frac{15}{16} \hat{\mu}_1 + |x|\right);
\]

\[
\hat{\mu}_2 \leftarrow \text{round}\left(\frac{15}{16} \hat{\mu}_2 + \min(4096, |x|^2)\right).
\]

We call $(\hat{\mu}_1, \hat{\mu}_2)$ the context of the current coefficient. To design a mapping from context to coding method, we collected all the wavelet coefficients belonging to each context using the Kodak lossless true color image suite and several photographs of cats collected from the Internet. We then found the best method for each context by brute force, compressing the coefficients using both interleaved and signed codes with power-of-two Golomb parameters ranging from 1 through 16. After inspecting the distribution of best coding methods over the context landscape, we designed a partitioning of the landscape using quadratic boundaries, described in Algorithm 1. We also use quadratic boundaries on $(\hat{\mu}_1, \hat{\mu}_2)$ to

![Fig. 1. Schematic of the neighborhood used to compute the context. The coefficient is shown as a black square, and the neighbors are shown as numbered squares (numbers represent relative weight), including eight neighbors in the same subband, two neighbors in the neighboring subbands, and one neighbor in the parent band. Only neighbors that precede the coefficient in scanline order may be used. For clarity, we show the wavelet decomposition in Mallat configuration, though our implementation uses an in-place transform.](image-url)
slower context method from (5), which is still substantially
than other transforms we tried. We employed the somewhat
versions of the same. For all color images, we employed the
library [1] for the original color images and for grayscale
compression results for our method vs. the JasPer JPEG 2000
of cats collected from the Internet [4]. Table 1 lists lossless
suite [3] and a large image comprising several photographs
evaluated our method on the Kodak lossless true color image
the code as it is released under the 3 clause BSD license. We
the code at
http://www.gfwx.org
besides standard C++ libraries. We invite the reader to view
lines of C++11 in a single header file with no dependencies
using C++11 templates, to facilitate customizing the data
pression pipeline and corresponding decompression pipeline
reversing the steps. We implemented the proposed com-
structing the decompression pipeline is straightforward, by
Though we describe only the compression pipeline, con-
ploy advanced quantization schemes such as Trellis coding to
for which it is not designed. In future work, we could em-
and introduces problematic artifacts on Bayer pattern data,
preserves more fine detail, but also has more ringing artifacts,
pixel, comparing our proposed method vs. JPEG 2000 (best
Fig. 2 shows a region of an image compressed to 1 bit per
E5620 CPU with hyperthreading. All configurations of our
libraries, running on a dual quad-core 2.4 GHz Intel Xeon
large block size vs. small block size to take advantage of mul-
tithreading, vs. the OpenJPEG [2] and JasPer [1] JPEG 2000
images, showing both context methods and
Table 2 lists encoding and decoding times for our method on
the large image of cats, showing both context methods and
large block size vs. small block size to take advantage of mul-
ihreading, vs. the OpenJPEG [2] and JasPer [1] JPEG 2000
libraries, running on a dual quad-core 2.4 GHz Intel Xeon
E5620 CPU with hyperthreading. All configurations of our
proposed method are faster than JPEG 2000 on this image
(upto 17 times faster), while still producing a smaller size.
Fig. 2 shows a region of an image compressed to 1 bit per
pixel, comparing our proposed method vs. JPEG 2000 (best
viewed in high resolution electronic format). JPEG 2000
preserves more fine detail, but also has more ringing artifacts,
and introduces problematic artifacts on Bayer pattern data,
for which it is not designed. In future work, we could em-
ploy advanced quantization schemes such as Trellis coding to
further improve quality at low bit rates.

Table 1. Lossless compression results (bytes) for our method vs.
JPEG 2000. (Y) indicates a grayscale version of the image, using
only the luma channel. Best sizes are highlighted in bold.

<table>
<thead>
<tr>
<th>Image</th>
<th>Ours</th>
<th>JP2K</th>
<th>Ours(Y)</th>
<th>JP2K(Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kodak01</td>
<td>513,752</td>
<td>510,526</td>
<td>263,412</td>
<td>267,235</td>
</tr>
<tr>
<td>Kodak02</td>
<td>465,084</td>
<td>450,482</td>
<td>202,704</td>
<td>207,203</td>
</tr>
<tr>
<td>Kodak03</td>
<td>399,544</td>
<td>397,835</td>
<td>173,972</td>
<td>175,538</td>
</tr>
<tr>
<td>Kodak04</td>
<td>463,872</td>
<td>460,159</td>
<td>202,332</td>
<td>206,711</td>
</tr>
<tr>
<td>Kodak05</td>
<td>532,040</td>
<td>531,776</td>
<td>255,936</td>
<td>260,523</td>
</tr>
<tr>
<td>Kodak06</td>
<td>475,884</td>
<td>471,536</td>
<td>231,788</td>
<td>233,317</td>
</tr>
<tr>
<td>Kodak07</td>
<td>419,540</td>
<td>418,095</td>
<td>184,804</td>
<td>185,476</td>
</tr>
<tr>
<td>Kodak08</td>
<td>554,260</td>
<td>547,613</td>
<td>269,668</td>
<td>271,490</td>
</tr>
<tr>
<td>Kodak09</td>
<td>454,500</td>
<td>445,098</td>
<td>195,764</td>
<td>196,662</td>
</tr>
<tr>
<td>Kodak10</td>
<td>459,748</td>
<td>453,211</td>
<td>198,564</td>
<td>201,113</td>
</tr>
<tr>
<td>Kodak11</td>
<td>460,076</td>
<td>456,826</td>
<td>220,736</td>
<td>223,955</td>
</tr>
<tr>
<td>Kodak12</td>
<td>423,244</td>
<td>425,654</td>
<td>192,124</td>
<td>193,588</td>
</tr>
<tr>
<td>Kodak13</td>
<td>581,164</td>
<td>583,094</td>
<td>293,760</td>
<td>300,050</td>
</tr>
<tr>
<td>Kodak14</td>
<td>500,804</td>
<td>499,524</td>
<td>243,224</td>
<td>247,866</td>
</tr>
<tr>
<td>Kodak15</td>
<td>440,072</td>
<td>442,318</td>
<td>192,404</td>
<td>195,095</td>
</tr>
<tr>
<td>Kodak16</td>
<td>433,860</td>
<td>431,410</td>
<td>203,664</td>
<td>205,678</td>
</tr>
<tr>
<td>Kodak17</td>
<td>453,940</td>
<td>451,947</td>
<td>202,220</td>
<td>207,277</td>
</tr>
<tr>
<td>Kodak18</td>
<td>549,572</td>
<td>546,925</td>
<td>247,108</td>
<td>252,670</td>
</tr>
<tr>
<td>Kodak19</td>
<td>493,036</td>
<td>482,870</td>
<td>221,572</td>
<td>223,442</td>
</tr>
<tr>
<td>Kodak20</td>
<td>405,004</td>
<td>397,111</td>
<td>180,740</td>
<td>181,948</td>
</tr>
<tr>
<td>Kodak21</td>
<td>487,132</td>
<td>479,938</td>
<td>225,624</td>
<td>228,514</td>
</tr>
<tr>
<td>Kodak22</td>
<td>503,516</td>
<td>496,250</td>
<td>223,064</td>
<td>226,958</td>
</tr>
<tr>
<td>Kodak23</td>
<td>421,288</td>
<td>418,135</td>
<td>171,516</td>
<td>173,523</td>
</tr>
<tr>
<td>Kodak24</td>
<td>497,996</td>
<td>499,855</td>
<td>231,464</td>
<td>233,523</td>
</tr>
<tr>
<td>Cats</td>
<td>17,838,708</td>
<td>18,440,076</td>
<td>7,818,056</td>
<td>7,875,114</td>
</tr>
</tbody>
</table>

Table 2. Lossless timings on the large image of cats.

<table>
<thead>
<tr>
<th>Method</th>
<th>Size (bytes)</th>
<th>Encode Time</th>
<th>Decode Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours (slow, large block)</td>
<td>17,838,708</td>
<td>5.1s</td>
<td>6.1s</td>
</tr>
<tr>
<td>Ours (slow, small block)</td>
<td>17,931,684</td>
<td>1.0s</td>
<td>1.2s</td>
</tr>
<tr>
<td>Ours (fast, large block)</td>
<td>18,276,476</td>
<td>2.2s</td>
<td>3.1s</td>
</tr>
<tr>
<td>Ours (fast, small block)</td>
<td>18,371,604</td>
<td>0.6s</td>
<td>0.7s</td>
</tr>
<tr>
<td>OpenJPEG</td>
<td>18,440,016</td>
<td>14.0s</td>
<td>12.0s</td>
</tr>
<tr>
<td>JasPer</td>
<td>18,440,076</td>
<td>10.6s</td>
<td>9.1s</td>
</tr>
</tbody>
</table>
7. ACKNOWLEDGEMENTS

This work was sponsored by the U.S. Army Research Laboratory (ARL) under contract number W911NF-14-D-0005, and the USC Shoah Foundation.

8. REFERENCES


