High efficiency lossless image recompression algorithm with asymmetric numeral systems for real-time mobile application

Qinghua Sheng
the School of Electronics and Information, Hangzhou Dianzi University

Haonan Zhu
the School of Electronics and Information, Hangzhou Dianzi University

Haixiang Sheng
the School of Electronics and Information, Hangzhou Dianzi University

Xiaofang Huang
the school of Electronics and Information, Hangzhou Dianzi University Information Engineering College

Jie Jiang
the School of Electronics and Information, Hangzhou Dianzi University

Changcai Lai (cclai@hdu.edu.cn)
the School of Electronics and Information, Hangzhou Dianzi University

Research Article

Keywords: Image compression, lossless, Asymmetric numeral systems, cache-friendly

Posted Date: January 11th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2455605/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Journal of Real-Time Image Processing on July 22nd, 2023. See the published version at https://doi.org/10.1007/s11554-023-01346-z.
High efficiency lossless image recompression algorithm with asymmetric numeral systems for real-time mobile application

Qinghua Sheng¹ · Haonan Zhu¹ · Haixiang Sheng¹ · Xiaofang Huang² · Jie Jiang¹ · Changcai Lai¹

Abstract JPEG images are widely used by individual users, data centers, cloud storages and network file systems. The huge transmission and storage bandwidth of existing JPEG images is becoming a big challenge due to its low compression efficiency. Some methods such as Google’s Brunsli can further compress these JPEG images adequately and restore them back to JPEG format losslessly when needed, but its decoding speed is not fast enough especially for real-time mobile applications. To further reduce its decoding complexity, we proposed three optimizations based on Brunsli, which include using asymmetric numeral systems coding to replace the existing arithmetic coding, designing the joint encoding of multiple symbols and cache-friendly optimizing the data structures. Experimental results demonstrate that compared with Brunsli, the proposed method achieves average 2 to 2.4 times decoding faster on mobile platform with comparable compression efficiency which is really valuable for power sensitive real-time mobile applications.

Keywords Image compression · lossless · Asymmetric numeral systems · cache-friendly

1 Introduction

As the world moves forward into the digital age, our reliance on electronic media is growing exponentially each year. According to Cisco’s Annual Internet Report [1], digital media accounts for approximately 80% of the world’s Internet traffic, and it is still growing. The statistics in Data never sleeps shows that the total amount of data consumed globally in 2021 was 79 zettabytes, an annual number projected to grow to over 180 zettabytes by 2025 [2]. Nowadays, images are widely used as an effective information transmission medium. Compared to words, pictures are more easily accepted by the brain and contain a wealth of information that is more easily articulated and transferred. According to Dropbox statistics, JPEG photos make for around 35% of the data stored by individual users in the cloud [3]. For mobile users, a large number of images take up a high amount of memory space. Therefore, for both individual users and businesses, how to store these images efficiently and transfer them quickly over the network is a significant challenge.

Developing powerful hardware, optimizing software or a combination of them are the three likely solutions to this issue [2]. However, the hardware solution is limited due to its cost and development cycle as reported in [4]. Hence optimized compression software is more straightforward to solve this problem by radically reducing the size of the source file and thus the bandwidth required for transmission. Digital images contain various redundancies, such as spatial redundancy, spectral redundancy, etc., which bring opportunities for image compression [5]. Image compression is divided into lossy and lossless compression according to whether or not the compressed file can be restored to the exact same original image. Compared with lossless compression, lossy compression can bring higher compression ratio, but the compressed file cannot restore the same original image. Some applications that need high quality images such as medical, geological exploration, etc.
require lossless compression as a way to compress images [6].

JPEG [7] is the most common format for storing and transferring photos on various browsers, and many mobile devices have CPUs that support hard encoding and hard decoding of JPEG. There is a large amount of JPEG images, both on the cloud and on individual phones, which takes up a lot of storage and also increases transmission bandwidth. We focus on lossless compression of JPEG images for the following reasons: First, in many application scenarios, it is necessary to compress and decompress images more than once, and if lossy compression is used, the images will be distorted after repeated compression and decompression, and the quality of the final recovered images will be significantly reduced. Second, considering the specific requirements of some users, they do not want to lose their image data due to privacy and data integrity, and require full recovery of the original data after decompression. Finally, storage space on personal mobile devices and in the cloud would be greatly reduced if lossless compression could be applied to large numbers of JPEG images.

Since the development of JPEG format, there are many algorithms with better performance than JPEG, such as JPEG XR [8], JPEG2000 [9], HEIF [10], intra frame compression of VVC [11], etc. In the era of rapid development of artificial intelligence, some methods [12–15] using deep learning methods to achieve image coding and decoding have also emerged. The purpose of these algorithms is to compress the original image to generate a corresponding format such as JPEG. However, most of the images saved on the web and on personal platforms are in JPEG format images and the above algorithms cannot losslessly compress existing JPEG format images. In order to recompress JPEG images, many algorithms such as Lepton [3], PackJPG [16], JPEG XL [17,18], Brunsli [19] have been developed.

Lepton, proposed by Horn et al., achieves a 22% reduction in storage after lossless recompression [20]. Lepton employs efficient binary arithmetic coding and considerably widens the probability models to achieve large compression gains, however this comes at a high cost in terms of storage and computation. When encoding the DC (Direct Current) component, lepton uses its AC (Alternating Current) component to predict and finally encodes the residuals.

Packjpg is another lossless compression method for JPEG pictures that can reduce storage requirements by 15%. Packjpg encodes an image according to the position of the coefficients, for a JPEG image, all DC coefficients are encoded together, and it needs to put all relevant coefficients into the same context, which requires a lot of time in decoding process [21].

JPEG XL, a flexible compression technique that supports both lossless and lossy compression. It is also possible to losslessly convert existing JPEG images to JPEG XL. The quantization matrix in JEPG XL may be adjusted locally to better match the complexity of various sections rather than being utilized globally at a fixed size. In addition, Huffman coding is replaced with ANS (Asymmetric Numeral Systems) [22] in JPEG XL. In contrast to the original DC coefficient prediction mode in JPEG, JPEG XL supports eight modes and will select the mode that produces the least error.

Brunsli is a lossless JPEG repacking library that uses a mixture of ANS and arithmetic coding in place of Huffman coding to compress JPEG images. Then pack them into specific format files while reducing the size by 22%. In addition, it can restore the compressed files to original JPEG images byte by byte. Compared with other lossless compression algorithms, it has the fastest decoding speed with little difference in compression ratio.

While data compression can reduce network bandwidth and storage capacity, it also requires computational resources for data compression and decompression, and this additional processing can be detrimental in applications such as embedded devices which have very limited processing power [23]. All of the above methods have impressive compression ratios, but the decoding speed is not fast enough especially on embedded devices. After experimental tests and comparisons, Google’s Brunsli project has a comparable compression rate to other algorithms, and has the best complexity performance in the process of decompression, but the decoding speed still cannot meet the requirements of some power sensitive real-time mobile applications.

To satisfy the requirements of real-time decoding, based on the Brunsli, we aim to propose a method that has a comparable compression rate to the above methods and can be decoded quickly on embedded devices with low processing power. In conclusion, our main contributions include:

1. Consider that ANS coding and arithmetic coding have similar compression ratio and ANS decoding speed is much higher than arithmetic coding. Based on the Brunsli project, the arithmetic encoding used by Brunsli is replaced by redesigned ANS encoding.
2. Combine Brunsli multiple binary code elements into a new code element and construct a contextual model that fits it. It enables to reduce the number of decoding and encoding times with small compression ability loss.
3. Further design the key data structure of ANS decoding in Brunsli to improve its cache friendly ability. The
large chunk of memory previously stored is replaced with simple arithmetic operations. Although the number of operations is slightly increased, the access memory is well optimized, with the result of a greatly increased cache hit rate, and shortened decoding time.

The rest of this paper is organized as follows: Section 2 briefly introduces the jpeg lossless recompression method of Brunsli. Section 3 describes our proposed schemes in detail. Experimental results and comparisons are given in Section 4, and section 5 concludes this paper.

2 Introduction of JPEG lossless recompression

As shown in Fig. 1, Google’s Brunsli library contains two parts: encoding (cbrunsli) and decoding (dbrunsli). During the encoding process, the JPEG image to be compressed is fed into the encoder as input for lossless encoding, firstly the JPEG standard decoding module decodes the input image to get the original JPEG coefficients, then the lossless encoding module encodes the coefficients into a code stream, finally the code stream is written to compressed file. In the process of decoding, the decoder obtains the specific format file as input, and parses the code stream into JPEG file header information and original JPEG coefficients. Then the JPEG standard encoding module encodes the coefficients and generates a JPEG picture, at which time the compressed file is restored to a JPEG image.

Fig. 1 Flowchart of the Google’s Brunsli scheme

In the Brunsli scheme, the smallest unit of encoding and decoding is an 8 × 8 sub-block. While encoding a sub-block, the last non-zero position in the 8 × 8 sub-block is firstly obtained and written to the code stream using arithmetic encoding so that only the previous coefficients need to be encoded in the subsequent encoding process. For a single coefficient, the Brunsli project will split a coefficient into several code elements. Firstly, it will determine whether the current coefficient is zero and encode it into the stream using arithmetic coding, secondly, if the coefficient is not zero, it will split this coefficient into sign bit and its absolute value. The sign bit is encoded using arithmetic coding, and the absolute value is encoded with a mixture of ANS and arithmetic coding. The decoding process corresponding to the encoding is shown in Fig. 2.

![Fig. 2 Decoding 8 × 8 block in Brunsli](image)

Although the main idea of Brunsli is beneficial to the compression rate with moderate decode complexity, the complicated arithmetic process and related conditional judgments are not friendly to real-time decoding in mobile applications, especially for images with large resolution. To address the existing problems with Brunsli, we propose to further optimize the flow of encoding and decoding to obviously reduce the decoding time.

3 Proposed methods

In the proposed lossless compression scheme, the design of encoding a sub-block is substantially simplified, which includes context prediction, model updating, and ANS encoding. The flowchart of the simplified encoder is shown in Fig. 3. First, the image after DCT transformation and quantization is fed into the encoder as input. Due to the great correlation between adjacent pixels, the information of the pixels adjacent to the current symbol to be encoded and the position information are used to compute the context of the current symbol. This
allows the distribution of symbols in the same probability space to be as inhomogeneous as possible which is beneficial for ANS to reduce the average code length later. In the next step, the to-be-encoded symbols are updated into the specified probability space which is aligned to the context computed in the previous step, and update the context model in order to accurately compute the next to-be-encoded symbol. Once all the symbols have been accessed, the symbols together with their probability space are transferred into the ANS encoder for encoding. Finally, the output of the ANS encoder and the probability space will be fed into the compressed bit stream.

3.1 Asymmetric numeral systems coding

To better introduce the proposed method, a brief description of the ANS is given firstly. rANS (range Asymmetric Number System) and tANS (table Asymmetric Number System) are two algorithms derived from ANS and they are tailored to different usage needs. rANS requires some arithmetic operations with higher complexity, such as multiplying integers, which take more clocks than addition, thus rANS is a relatively higher complexity implementation of ANS. tANS converts all arithmetic operations into a table lookup operation, i.e. tANS saves all possible calculations in a table in advance and then looks them up when a calculation needs to be performed. However, for tANS, the size of the table needs to be very large to obtain a comparable compression rate, which is not possible for some devices with scarce memory space resources, such as embedded devices [24]. Since the scheme will eventually be implemented on an embedded device, the limitation of memory resources is a fatal problem compared to its limited computational resources. Therefore, rANS is selected as a backbone entropy coding approach in our proposed algorithm.

Suppose that the set of symbols $A(a_1, a_2, \cdots, a_j)$, where each element $a_j$ is a source symbol, and $a_j$ is given by the distribution $p_j$, then the average information output for this source is shown in eq. (1).

$$H(X) = - \sum_{i=1}^{j} p_i \times \log_2 p_i$$  \hspace{1cm} (1)

$$\sum_{i=1}^{j} p_i = 1$$

Where $H(X)$ is the average amount of information for this source, or named as the entropy of the source. ANS uses a single natural number $x \in N$ as the state, suppose that some string $b_1b_2\cdots b_i$ have been already converted into a natural number $x_i$ by the conversion function $C_{opt}$. If another symbol $b_{i+1}$ is obtained, and a function is derived such that $x_{i+1} = C_{opt}(x_i, b_{i+1})$, it generates the representation of $b_1b_2\cdots b_{i+1}$. This encoding function must satisfy the rule that the number encoded comes out as close as possible to the entropy of the original string. Assuming that the probability of $b_{i+1}$ is $p_{i+1}$, according to the knowledge of information theory that the amount of self-information carried by $b_{i+1}$ is $-\log_2 (p_{i+1})$ and $b_{i+1}$ can be encoded optimally with $-\log_2 (p_{i+1})$ bits, the optimal encoding function can be derived as the following formula:

$$\begin{align*}
H(x_{i+1}) &= H(C_{opt}(x_i, b_{i+1})) \\
&= H(x_i) + H(b_{i+1}) \\
&= \log_2 (x_i) - \log_2 (p_{i+1}) \\
&= \log_2 \left( \frac{x_i}{p_{i+1}} \right)
\end{align*}$$

(2)

According to the above equation, if the probabilities of symbols are known, the compressed bits stream that approximates the compression limit can be obtained. In addition, due to the presence of fractional division in the above equation which is very complicated to calculate, rANS is used to encode multi-symbol which works on the same principle as the above-derived formula. The method normalizes the symbol probabilities so that the symbols are assigned to the range of $2^n$ which converts the original division operation into an integer division and a shift operation. And the symbols are distributed...
in range of 0 to $2^n$ according to their probability as shown in Fig. 4. Finally, the above encoding function is converted to the following form:

$$C(x_i, s_{i+1}) = \left\lfloor \frac{x_i}{f_s} \right\rfloor \times 2^n + CDF[s] + (x_i \mod f_s)$$

$$CDF[s] = f_0 + f_1 + \cdots + f_{s-1}$$

And the corresponding decoding function is shown in eq. (4).

$$s_{i+1} = \text{symbol}(x_{i+1} \mod 2^n) \text{ such that } CDF[s] \leq x_{i+1} \mod 2^n < CDF[s+1]$$

$$x_i = D(x_{i+1}) = f_s \times \left\lfloor \frac{x_{i+1}}{2^n} \right\rfloor - CDF[s] + (x_{i+1} \mod 2^n)$$

From the above encoding and decoding equations, the symbols can be encoded and decoded using the above equations as long as the probability distribution of each symbol is known in advance. Therefore, before encoding and decoding, the related symbols need to be pushed into their corresponding probability space for the encoding and decoding operation.

Fig. 4 Distribution of “yellow”, “green” and “blue” symbol

### 3.2 Joint encoding of multiple symbols

Considering the complex fractional operation of arithmetic coding, much related work has tried to improve and optimize it such as video coding, where the binary arithmetic coding converts the multiplication and division operations on floating-point numbers to shifting and multiplying integers, which greatly improves the coding speed. However, the input parameters of the binary arithmetic encoder must be 0/1, thus it is necessary to binarize the encoded symbols before encoding, and the original symbol has to split into multiple binary symbols, which leads to an increase in the number of encoding operation. In Brunsli, its binarization operation is performed on elements such as whether the block is empty or not, whether the current code element to be encoded is zero, and the sign of the code element. In this paper, taking advantage of the fact that ANS is faster than arithmetic encoding in multi-symbol encoding, these three code elements are combined into one symbol. Ideally, using just one encoding operation instead of three arithmetic encodings which can optimize the encoding speed.

Since there is a correlation between the encoded symbol and the symbol to be encoded, the previously encoded information (i.e., the context) can be used to conditionally encode the symbol according to the theory of conditional entropy. Suppose the symbol to be encoded is $X$ and it has a priori information $C_1C_2\ldots C_N$, then its conditional entropy can be derived as eq. (5).

$$H(X \mid C_1, \ldots, C_{N-1}) = -\sum_{i=1}^{q} \sum_{i=N}^{q} p(C_{i1}, \ldots, C_{iN}) \log p(X \mid C_{i1}, \ldots, C_{iN})$$

According to the paper [25], the more adequate a priori information is known, the lower entropies can be achieved. However, the more parameters the model has, the more training data are needed to train the optimized model, otherwise overfitting problems may occur, which could result in a decrease in compression efficiency. For a symbol to be encoded, its reference information can be shown as follows.

1. information about the above and previous adjacent symbols
2. information about the position of the current symbol

As seen from Fig. 5, $I(x, y)$ represents the current pixel value at the position $(x, y)$. In this step, the gray values of the above and left pixel of input image are used to compute the context of $I(x, y)$. Since the symbols to be encoded are the data of original image after quantization of DCT transformation coefficients, processed by the scan order shown in Fig. 6. The probability of occurrence of zero in the lower right corner is greatly increased after the zig-zag scan, hence the information about the position of the current pixel in an 8*8 sub-block is also an important factor that affects the context’s precision. In our design, in order to accommodate the joint coding of multiple code elements while reducing the complexity of the model, the conditions associated with them are combined. And the independence between each condition is guaranteed to ensure that each condition is fully utilized. The context is computed by the equation as shown in eq. (6).

$$\text{context} = (I(x, y-1) + I(x-1, y)) \times \text{pos} + \text{pos}$$

5
3.3 Optimizing cache-friendly data structures

To mitigate the effects of the growing gap between processor speed and memory access speed, today’s computer architectures employ hierarchical memory architectures shown in Fig. 7. The cache was created to resolve the mismatch between the CPU’s computing speed and the memory’s read/write speed because the CPU’s computing speed is much faster than the memory’s read/write speed, which makes the CPU spend a long time waiting for data to arrive or writing data into memory [26]. Cache memory is a temporary memory that sits between the CPU and the memory, its capacity is much smaller than the memory but its swap speed is much faster than the memory. If the cache hits, the data will be transferred directly from the cache to the CPU without going through a slow access to main memory. If the cache does not hit, the data is read from memory at a relatively slow speed and sent to the CPU for processing, and the block in which this data is located is transferred to the cache. This operation makes it possible to read the entire block of data from the cache for subsequent adjacent instructions, without having to access memory again.

On the decoding side, the most frequently accessed data are the probability space of the code element and the symbol array of ANS. Here two ways are proposed to get a higher cache hit rate. Compared to the Google Brunsli project, its ANS-related array adopts a space-for-time approach, i.e., the frequency of symbols, the cumulative probability of symbols, and the type of symbols are combined into one data structure as shown in Algorithm 1 which has the advantage of eliminating a modulo operation during the decoding calculation. The storage space required for a histogram in Brunsli is $1024 \times (2B + 2B + 2B) = 6KB$ and the corresponding decoding function calculation is given by eq. (7). In our design, this data structure only has the start of cumulative probability, and its frequency as shown in Algorithm 2, which greatly reduces the size of the space occupied by the symbol array. After data structure optimizing the storage space decreases to $18 \times (2B + 2B) + 1024 \times 1B = 1096B$ and the decoding function calculation is shown in eq. (8). Although there is an additional modulo operation in the calculation, the cache hit rate is greatly improved because the space of the array becomes smaller. According to experimental tests, the optimization of this data structure is beneficial to reduce the decoding time.

![Fig. 5 Prediction template of the proposed method](image)

![Fig. 6 Zig-zag scan order](image)

![Fig. 7 CPU Architecture-Cache](image)

**Algorithm 1** ANS encoding step of symbol s from state x

1: struct {
2: uint16 offset
3: uint16 freq
4: uint8 symbol
5: } ANSSymbolInfo ANS
6: Corresponding decoding function from state $x_i$
7: ANSSymbolInfo info=map[$x_i$]
8: uint8 s=info.symbol
9: uint16 f=info.freq
10: uint16 o=info.offset $ANS\_LOG\_TAB\_SIZE=n$

$$x_{i+1} = f \times x_i \gg ANS\_LOG\_TAB\_SIZE + o \quad (7)$$

**Algorithm 2** ANS encoding step of symbol s from state x

1: struct {
2: uint16 start
3: uint16 freq
4: } ANSSymbolInfo
5: ANS
6: ANS\_MAX\_SYMBOL=the num of symbols
7: ANSSymbolInfo map[ANS\_MAX\_SYMBOL]
8: Corresponding decoding function from state $x_i$
9: ANSSymbolInfo info=map[$x_i$]
10: uint8 s=map[info]
11: ANSSymbolInfo info=map[$s$]
12: uint32 f=info.freq

$$x_{i+1} = f \times (x_i \gg ANS\_LOG\_TAB\_SIZE) + (\text{Mod}(x_i, ANS\_TAB\_SIZE)) - \text{start} \quad (8)$$
Another way to optimize cache hit rate is to classify the probability space of each code element. With the guarantee that the individual probability spaces do not change much, a merging operation is performed on them so that their distribution of the same type is as compact as possible. When the decoding of one symbol is completed, there is a higher probability that the probability space required for the next code element will be available in the cache. This step brings a small loss of compression rate, but it is acceptable by considering the speed optimization that it brings.

### 4 Experimental results and analysis

Lossless image compression algorithm is usually evaluated by the following parameters, compression ratio (CR), encoding time (ET) and decoding time (DT). The importance of the three parameters should be adjusted for various applications while evaluating the algorithm’s benefits [2]. Since the purpose of proposed solution is to be implemented on a real-time embedded platform, and most embedded devices have a small storage space, so the compression ratio is an important metric. In addition, due to the limited power of embedded devices and the decode operation being more frequent than encode, the decoding time counts from clicking on the compressed file to the appearance of the picture on the screen, is extremely important to the user experience. It is worth mentioning that the encoding time is less important than the decoding time, mainly because the user can encode the image at their free time and generally the encode is done not many times. Therefore, when evaluating the effectiveness of the algorithm in this paper, we use decoding speed and compression ratio as two indicators. For the speed test, the disk I/O time is ignored because it is not relevant to the algorithm optimization. The compression ratio performance is measured by the bit-saving of the compressed bitstream (with a size of $S_c$) compared to the size of original JPEG files $S_o$. The corresponding derivation is given by eq. (9).

$$\text{compression rate saving} = \left(1 - \frac{S_c}{S_o}\right) \times 100\% \quad (9)$$

### Datasets

The dataset used in the experiments consists of two parts of images, the large part are daily photos taken by mobile phones and another part obtained from digital camera photos. Due to the low computing power of the embedded device, a small perturbation during the experiment will have a large impact on the performance test. Therefore, in this test, the size of images used range from 1M to 15M, which increases the decoding time and improves the accuracy and precision of experimental measurements. In addition, the measurements are repeated for the same set of images several times and finally take the average value to reduce the experimental error. Today’s smartphones are becoming more and more powerful in taking pictures with the development of camera technology, the number of pixels in a single picture is increasing. The majority file size of pictures taken in our daily life fall within the range of 1MB-15MB, so this dataset is universal for image compression test. Images are divided into five collections according to their size. For example, collection A contains photos whose size is in the range of 2MB-4MB and collection E contains photos whose size is in the range of 10MB-12MB.

### Implementation details

In the experiments, mobile phone is the main test platform, a X86-64 based server is also used as a comparison assist test platform. The anchor algorithms on the server consist of Brunsli, Lpeton, and Packjpg. Some algorithms make use of multi-threading for performance optimization, while others do not, thus all algorithms in the test are run in single-threaded mode for fair comparison. The server testing results are obtained on a PC with an Intel Core i9-10900K CPU (3.70GHZ) and 32GB RAM. It’s found in the test that among the above algorithms, the compression rate of Brunsli is comparable to the other algorithms and the decoding speed is much better than the other algorithms. Therefore, on the embedded side, the best performing Brunsli is used as anchor in experiment. The embedded platform testing results are conducted on HUAWEI Mate9 equipped with Hisilicon Kirin 960 and 4GB RAM.

### Overall Performance

Table 1 shows the compression results on server for all the five collections. The bit-saving of proposed scheme ranges from 20.19% to 21.85% on different image collections, and the average bit-saving is 20.71% which is comparable to others. The left of Fig. 8 shows the compression efficiency of four different algorithms. The right one shows the decompression pression time of proposed scheme in comparison with some other JPEG recompression methods. Among the four compression schemes, the proposed scheme achieves the best performance in decompression speed. The decompression speed of proposed scheme outperforms Brunsli, Lpeton and Packjpg by 1.61, 2.63, 6.31 times respectively.

On the embedded platform, Brunsli which has the best performance is chosen as anchor. Fig. 9 shows decompression time of proposed scheme and Brunsli on mobile phone. Since the change of the experimental platform can only change the running time of the algo-
Table 1: Overall performance of Brunsli, Lpeton, Packjpg and ours on server

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>22.96</td>
<td>23.05</td>
<td>23.32</td>
<td>22.98</td>
<td>24.65</td>
</tr>
<tr>
<td>Lpeton</td>
<td>22.33</td>
<td>22.8</td>
<td>22.94</td>
<td>22.76</td>
<td>24.33</td>
</tr>
<tr>
<td>Packjpg</td>
<td>22.32</td>
<td>22.63</td>
<td>22.9</td>
<td>22.78</td>
<td>24.28</td>
</tr>
<tr>
<td>Proposed</td>
<td>20.3</td>
<td>20.19</td>
<td>20.48</td>
<td>20.7</td>
<td>21.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>0.36</td>
<td>0.403</td>
<td>0.473</td>
<td>0.492</td>
<td>0.518</td>
</tr>
<tr>
<td>Lpeton</td>
<td>0.592</td>
<td>0.66</td>
<td>0.761</td>
<td>0.791</td>
<td>0.825</td>
</tr>
<tr>
<td>Packjpg</td>
<td>1.655</td>
<td>1.844</td>
<td>1.791</td>
<td>1.802</td>
<td>2.517</td>
</tr>
<tr>
<td>Proposed</td>
<td>0.232</td>
<td>0.257</td>
<td>0.288</td>
<td>0.289</td>
<td>0.307</td>
</tr>
</tbody>
</table>

Fig. 8: Compression rate-savings and decompression time of our scheme and other JPEG recompression methods on server.

Table 2: Overall performance of Brunsli and ours on embedded platform

<table>
<thead>
<tr>
<th>Collection</th>
<th>Decoding times (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Brunsli</td>
<td>0.642</td>
</tr>
<tr>
<td>Proposed</td>
<td>0.302</td>
</tr>
</tbody>
</table>

Fig. 9: Overall performance of Brunsli and ours on embedded platform.

Efficiency of the Asymmetric numeral systems coding

The efficiency of Asymmetric numeral systems coding is evaluated by whether to use ANS as a replacement for arithmetic codes in proposed scheme. Table 3 and Fig. 10 show the comparison results for all the five photo collections. It can be observed that the contribution of the Asymmetric numeral systems coding varies for different collections. Collection D and E show the best performance. The main reason is that for images with fewer pixels, the final stage of ANS coding is a smaller part of the overall process and its optimization is less effective than images with more pixels. Another reason is that the decoding time of the small images is short and a little fluctuation during the test can have an impact on the results, although the statistical averaging method of measuring multiple groups is used to reduce the measurement error.

The result shows that due to the replacement of the arithmetic code with ANS the compression ratio is reduced by an average of 2.1%, but the decoding speed is improved by about 1.2 times on different image collections. It is worth mentioning that although this step has a small loss on compression ability, it is necessary as the ANS replacement arithmetic code is the basis for the subsequent optimizations.

Efficiency of the Joint encoding of multiple symbols

Table 4 and Fig. 11 show the efficiency of Joint encoding of multiple symbols for all the five photo collections. The result indicates that the algorithm with joint encoding of multiple symbols reduces the compression...
Table 3 Efficiency of Asymmetric numeral systems coding

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>23.11</td>
<td>24.07</td>
<td>22.58</td>
<td>24.65</td>
<td>19.72</td>
</tr>
<tr>
<td>proposed</td>
<td>21.3</td>
<td>21.61</td>
<td>20.96</td>
<td>22.56</td>
<td>17.2</td>
</tr>
<tr>
<td>Decoding times (s)</td>
<td>0.642</td>
<td>0.68</td>
<td>1.857</td>
<td>2.44</td>
<td>3.282</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Brunsli</td>
<td>0.575</td>
<td>0.611</td>
<td>1.64</td>
<td>2.004</td>
<td>2.62</td>
</tr>
<tr>
<td>proposed</td>
<td>0.517</td>
<td>0.543</td>
<td>1.502</td>
<td>1.887</td>
<td>2.366</td>
</tr>
</tbody>
</table>

Table 4 Efficiency of Joint encoding of multiple symbols

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>23.11</td>
<td>24.07</td>
<td>22.58</td>
<td>24.65</td>
<td>19.72</td>
</tr>
<tr>
<td>proposed</td>
<td>22.13</td>
<td>23.67</td>
<td>21.89</td>
<td>24.17</td>
<td>19.37</td>
</tr>
<tr>
<td>Decoding times (s)</td>
<td>0.642</td>
<td>0.68</td>
<td>1.857</td>
<td>2.44</td>
<td>3.282</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Brunsli</td>
<td>0.517</td>
<td>0.543</td>
<td>1.502</td>
<td>1.887</td>
<td>2.366</td>
</tr>
<tr>
<td>proposed</td>
<td>0.517</td>
<td>0.543</td>
<td>1.502</td>
<td>1.887</td>
<td>2.366</td>
</tr>
</tbody>
</table>

Fig. 10 Compression rate saving and decompression time with ("proposed") and without ("Brunsli") Asymmetric numeral systems coding.

Efficiency of the Optimizing cache-friendly data structures

Table 5 and Fig. 12 show the efficiency of cache-friendly data structures. It can be obtained that the compression rate of the cache-friendly data structures related algorithm is very close to the anchor. The optimization of the data structure converts the original large table lookup operations into simple arithmetic operations, eliminating the need for table lookups and reducing the size of the data structure. The slight reduction in compression rate is caused by the merging probability spaces which would produce a change in the probability ratio by an average of 0.58% compared to the anchor. This is mainly due to the fact that fewer conditions can be accessed to multiple code elements than multiple single code elements, and the conditional entropy of each code element becomes larger, resulting in an increase in the required code length of a single code symbol. It can be also observed that the decoding speed of the algorithm with joint encoding of multiple symbols is on average 1.38 times faster than Brunsli on different image collections. The reason is that the combination of multiple code elements reduces the total number of encodings, resulting in a faster decoding speed.
Table 5 Efficiency of cache-friendly data structures

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>23.11</td>
<td>24.07</td>
<td>22.58</td>
<td>24.65</td>
<td>19.72</td>
</tr>
<tr>
<td>proposed</td>
<td>22.97</td>
<td>23.97</td>
<td>22.47</td>
<td>24.54</td>
<td>19.57</td>
</tr>
</tbody>
</table>

Decoding times (s)

<table>
<thead>
<tr>
<th>Collection</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brunsli</td>
<td>0.642</td>
<td>0.68</td>
<td>1.857</td>
<td>2.44</td>
<td>3.282</td>
</tr>
<tr>
<td>proposed</td>
<td>0.52</td>
<td>0.522</td>
<td>1.288</td>
<td>1.862</td>
<td>2.515</td>
</tr>
</tbody>
</table>

Fig. 12 Compression rate saving and decompression time with (“proposed”) and without (“Brunsli”) cache-friendly data structures

bility of a symbol. The decoding speed of the algorithm with cache-friendly data structures is on average 1.34 times faster than Brunsli on different size of images.

5 Conclusions and future work

In this paper, we propose a new hybrid lossless image compression scheme combining Asymmetric numeral systems coding, the joint encoding of multiple symbols and cache-friendly data structures. The experimental results show that the proposed algorithm outperforms other state-of-art algorithms in terms of decoding speed, with a little loss of compression ability. When images become more complex in higher resolution, the performance of proposed method becomes better. It can greatly reduce the storage cost for backup and archive of JPEG coded image for both personal and cloud applications. When the user intends to view the image, the proposed solution can quickly decode the compressed images into JPEG files, which is important to the real-time mobile applications.

The proposed method mainly focuses on natural images with 256 colors. In the future, it is interesting to modify the method to work with a wider range of image types like color-indexed images with a small color palette. In terms of multi-code element unification, it is also important to find more adequate conditions for the current coding element, for the purpose of reducing its conditional entropy.

References