VaFLE: Value Flag Length Encoding for Images in a Multithreaded Environment

Bharath A. Kinnal(✉), Ujjwal Pasupulety, and V. Geetha

Department of Information Technology, National Institute of Technology Karnataka Surathkal, Mangaluru 575025, India
1997bhar@gmail.com,
{15it150.ujjwal,geethav}@nitk.edu.in

Abstract. The Run Length Encoding (RLE) algorithm substitutes long runs of identical symbols with the value of that symbol followed by the binary representation of the frequency of occurrences of that value. This lossless technique is effective for encoding images where many consecutive pixels have similar intensity values. One of the major problems of RLE for encoding runs of bits is that the encoded runs have their lengths represented as a fixed number of bits in order to simplify decoding. The number of bits assigned is equal to the number required to encode the maximum length run, which results in the addition of padding bits on runs whose lengths do not require as many bits for representation as the maximum length run. Due to this, the encoded output sometimes exceeds the size of the original input, especially for input data where in the runs can have a wide range of sizes. In this paper, we propose VaFLE, a general-purpose lossless data compression algorithm, where the number of bits allocated for representing the length of a given run is a function of the length of the run itself. The total size of an encoded run is independent of the maximum run length of the input data. In order to exploit the inherent data parallelism of RLE, VaFLE was also implemented in a multithreaded OpenMP environment. Our algorithm guarantees better compression rates of upto 3X more than standard RLE. The parallelized algorithm attains a speedup as high as 5X in grayscale and 4X in color images compared to the RLE approach.

Keywords: Run length encoding · Data parallelism · OpenMP · Lossless encoding · Compression ratio

1 Introduction

Data compression techniques have enabled networked computing systems to send larger amounts of data by effectively utilizing the channel bandwidth. Modern methods make use of complex algorithms to encode the original data bits such that the information they carry is preserved in a smaller number of bits. Data compression can either be lossy or lossless. Lossless compression ensures that the original data can be perfectly reconstructed from the encoded data. By contrast, lossy compression reconstructs only an approximation of the original data, though this usually improves compression rates.

A. B. Gani et al. (Eds.): ICICCT 2019, CCIS 1025, pp. 48–60, 2019.
https://doi.org/10.1007/978-981-15-1384-8_5

Run Length Encoding (RLE) is a simple and frequently used lossless encoding method. It involves scanning the input data and storing the input symbol and the length of the symbol run.

RLE is primarily used to compress data from audio, video, and image files. It works exceptionally well with black and white images as well as grayscale images. This is because the data in such files contains large length runs of contiguous 1s or 0s, yielding high compression rates.

One major issue with RLE is that the encoded runs have their lengths stored in a fixed number of bits in order to simplify decoding. This fixed number is equal to the bits required to represent the maximum length resulting in low encoding rates on highly varying input data. The proposed lossless VaFLE algorithm encodes the run lengths such that the number of bits allocated is an increasing function of the run length itself. Contrary to the traditional method of representing runs in the form of (Value, Length) pairs, VaFLE makes use of (Value, Flag, Length Offset) triples. Therefore, the total size of an encoded run independent of the maximum run length of the input data. The end result is an algorithm which offers high compression ratios at lower execution times, which can be seen by analysing the space complexity of VaFLE algorithm's effectiveness, and comparing to that of the RLE approach.

As RLE and RLE-based algorithms possess an inherent form of data parallelism, VaFLE was further optimized to run on multi-core systems by splitting the image file into a number of small chunks, and compressing each chunk in parallel across several processing cores. The OpenMP [4] environment was used to run data parallel operations on different cores, as well as integrating compressed segments in a sequential manner. The algorithm can be used on a variety of multimedia files since it is format independent, but works best on black and white images.

The paper is structured as follows: Sect. 2 provides details on the literature survey. Section 3 explains the novel serial and parallelized VaFLE algorithms along with the space complexity analysis. Section 4 provides details about experimental results. Section 5 demonstrates our results and analysis. The paper concludes in Sect. 6.

2 Literature Survey

2.1 Previous Work and Applications

The RLE data compression algorithm has been used in a wide area of applications. Vijayvargiya et al. [9] conducted a survey on various image compression techniques which included Run Length encoding as a popular method. Chakraborty and Banerjee [3] used RLE along with special character replacement to develop an efficient lossless colour image compression algorithm. Arif and Anand [2] found another useful application for RLE based compression for speech data. Khan et al. [6] applied RLE in the domain of steganography which has a variety of practical implications in the cyber-security space.

RLE and RLE-based algorithms inherently have the property of data parallelism which can be exploited at the programming level as well, without the need for dedicated hardware. This can be seen in the publication by Trein et al. [8] who created a hardware implementation for Run Length Encoding where images are transferred as blocks of pixels on operated upon in parallel.

2.2 Basics of Run Length Encoding

Run Length Encoding is one of the simplest lossless encoding techniques. RLE compresses a given run of symbols by taking the value of that symbol in the run, and places next to it the number of times the same symbol has occurred in the run consecutively. This significantly reduces the size of the sequence and the new sequence is easy to interpret. For example, consider the following sequence:

AABBCCCCDD

With RLE, the sequence reduces to:

A2B2C4D2

The new sequence does not contain as many bits as the previous sequence. However, this works only if there is one bit that occurs much more often in the sequence as the other. In case of equality, RLE proves to be highly inefficient as it would generate a sequence that would be double the size of the original sequence. For example, consider:

1010

On performing RLE, the following string would be obtained:

11011101

The output is exactly twice the size of the original sequence, which defeats the purpose of making the smaller string. RLE is useful when it comes to sequences that contain a large quantity of one bit and can be used for characters as well as bits. In the best case, RLE can reduce data to just two numbers if all the values in the original data are exactly the same, regardless of the size of the input. But in the worst case, i.e, if there are no repeating values in the data, RLE can double the size of the original data.

2.3 Modified RLE with Bit Stuffing

Performing RLE on smaller sequences actually results in the expansion of the data rather than compression. Amin et al. [1] devised a modified run length encoding scheme which addresses some drawbacks of the original RLE algorithm. This algorithm makes use of bit stuffing and avoids encoding small sequences of one or two bits. The input data is analyzed to highlight if there are any large sequences that will decrease the number of bits to represent the length of each run. The algorithm used in [1] incorporates bit stuffing by inserting non-information bits into the data. The location of stuffed bits is communicated to the receiving end of the data link, where they are removed to recover the original bit streams.

However, the concept of bit stuffing used in this algorithm may be counter-productive, as it could lead to the formation of longer runs. Furthermore, the maximum size for the representing the length of a given bit sequence is still a fixed value. The modified RLE algorithm makes use of bit stuffing to limit the number of consecutive bits of the same value in the data to be transmitted in an optimal manner. The strategy of ignoring small sequences was also incorporated. The combination of bit stuffing and ignoring small sequences results in a much more optimal Run Length Encoding Algorithm.

3 Value Flag Length Encoding

Both the original RLE and modified [1] schemes are based on finding the frequency of the sequential bits of a given value. One of the main problems observed is that the maximum size for representing the length of a given run is a fixed value. This constrains the space complexity of a given run of same-valued bits. The proposed VaFLE algorithm ensures the size required for representing length is a function of the original run length itself. This algorithm also ignores the runs of lengths equal to the minimum length specified by a variable called τ.

3.1 Serial VaFLE Algorithm

Algorithm 1 shows the basic Value Flag Length Encoding Scheme. The bit stream can be broken up into a sequence of multiple runs of same-valued bits. Only runs having a length greater than the minimum threshold length (specified by Threshold or τ) will be considered for analysis. In the algorithm a given run will be encoded in the form of a triple. The first part indicates the Value of the bits stored (exclusively 1s or exclusively 0s). The length of a given run is represented in the next two parts, namely the Flag and Offset. The Flag which specifies the number of bits required to encode the Offset.

Since Value should store a run of similar bits, its size should be equal to τ, as minimum as possible keeping in mind that it should resemble the sequence of bits of length less than τ. The Flag is indicative of the number of bits required to represent the offset. Consider a given run with a Value segment that contains bit b (exclusively 1 or 0) and the number of bits in the run is n. The Flag is set as k bits of value b, followed by a single bit which is the boolean complementary value of b. The value of k is found using (1).

$$\mathrm{k} = \log_2(n - (\tau - 1)) \tag{1}$$

For example, if value = ‘11’, then flag = ‘111….(k times) 0’. Here, 0 indicates the end of the flag and the start of the length offset. With the same value of k, the offset can be found. Given the original length n, the length offset is calculated using (2) for $n > \tau$.

$$\mathrm{lengthOffset} = (\mathrm{n} - 1) - 2^{\mathrm{k}} \tag{2}$$

Algorithm 1: Serial VaFLE Encoding

```
Input: Original file, and threshold length Threshold
Output: Compressed file
function VaFLE_encoding(file, Threshold)
    s1 ←* file
    s2 ←* compressedFile
    writeFile(s2, Threshold)
    size ←sizeof(file)
    iRun ← 0
    while iRun < size do
        count ← length of current run
        if count ≤ Threshold then
            writeFile(s2,bits($s1_{iRun}$, count))
        else
            value ←bits($s1_{iRun}$, Threshold)
            flag ← NULL
            flagSize ←$\log_2$(count − (Threshold −1))
            append(flag, intToBits($2^{flagSize \times s1\ iRun}$ − 1, flagSize))
            append(flag, !(bit($s1_{iRun}$)))
            length ← NULL
            lSize ← flagSize
            offset ← count - (Threshold - 1) - $2^{lSize}$
            append(length,intToBits(offset, lSize))
            writeFile(s2, value + flag + length)
        iRun ← iRun + count
    return compressedFile
```

Algorithm 2: Serial VaFLE Decoding

```
Input: Compressed file
Output: Original file
function VaFLE_decoding(compressedFile)
    s1 ←* compressedFile
    s2 ←* originalFile
    Threshold ← (int)extractHeader(s1)
    iRun ←sizeof(int)
    while iRun < sizeof(compressedFile) do
        runEnd ← iRun + (length of current run)
        if runEnd ≤ sizeof (compressedFile) then
            break
        if runEnd − iRun ≤ Threshold then
            writeFile(s2,bits($s1_{iRun}$, runEnd − iRun))
            iRun ← runEnd
        else
            offset ← runEnd − (iRun + Threshold + 1)
            tnum ←bitsToInt($s1_j$, j + offset − 1)
            num ← $2^{offset}$ + tnum + Threshold − 1
            writeFile(s2, intToBits($2^{num \times s1\ iRun}$ − 1, num))
            iRun ← j + offset
    return originalFile
```

To see how the algorithm runs, consider the following bit stream ‘0111111111111111’ and $\tau = 2$. If the size is not more than τ, it can be ignored. Thus the single 0 is encoded as it is. For the next run, the size, $n = 15$, which is greater than τ. This run can be encoded as a (Value, Flag, Offset) triplet. Being a sequence of 1s and the variable τ set to 2, the Value will be ‘11’. The value of k using (1) is equal to 3. Flag is given as k 1s followed by a 0. Next, using Eq. (2), the length offset is found to be equal to 6 whose binary representation is ‘110’ and requires only k bits to store.

Flag = ‘1110’
Length = ‘110’

Here, it must be noted that the offset must be of length = k, in order to decode the offset as that of a specified length given by the flag. The triplet is given as

(11,1110,110)

Thus the encoded data would be

Encoded data : 0, (11,1110,110)
Final bit string : ‘0111110110’

The length of the encoded bit string is 10. For the given example, the compression ratio is equal to 1.6. Algorithm 1 demonstrates the sequential encoding VaFLE scheme.

Decoding of the encoded bit string, shown in Algorithm 2, can be done by observing the value and finding the length with flag and length offset. To check if a run is encoded into a triplet, the first m consecutive bits having the same bit value should be observed. If $m \leq \tau$, then no change has to be made to run, and it is directly appended to the output. If not, then the length is found by looking at the flag and length offset.

3.2 Complexity Analysis of VaFLE

For runs of length $n \leq \tau$, it is assured that they will remain unchanged in the encoded string, and the encoded size equals n. For runs of length $n > \tau$, the total length of the encoded run of bits can be found by calculating the lengths of individual elements in the triplet. The value will have a constant length of τ bits.

$$v = \tau \tag{3}$$

Deriving the value of k from (1), the flag will have the following length,

$$f = 1 + k \tag{4}$$

The length offset lies in the range of $(n - 1) - 2^k$, where $n \in [2^k, 2^{(k+1)} - 1]$. There can be a total of 2^k values. These values can be encoded using k bits.

$$l = k \tag{5}$$

Combining (3), (4), (5), and (6) and substituting k using (1) the compressed size is calculated as follows,

$$\begin{aligned} s &= v + f + l \\ &= \tau + (k + 1) + k \\ &= \tau + 1 + 2(\log_2(n - (\tau - 1))) \end{aligned} \qquad (6)$$

Thus, the value of the length s is found to be a function of n, which shows that the length of the encoded run is dependent only on that run length alone, and not on the length of any other run.

As both the encoding and decoding algorithms involve going through all the bits in a sequential manner, the overall time complexity for both algorithms is of the order O(n), where n is the number of bits in the file being encoded or decoded.

3.3 Parallelized VaFLE in a Multithreaded Environment

While VaFLE on its own is more efficient than the naive RLE encoding scheme, input data is still scanned in a sequential manner. The algorithm has a time complexity = O(n). A parallel implementation would involve splitting up the input data and compressing each part separately. Data parallelism is a form of parallelization across multiple processing cores in parallel computing environments. It focuses on distributing data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel.

Algorithm 3 shows the parallel Value Flag Length Encoding Scheme. The inherent data parallelism of RLE algorithms has been exploited in order to create a parallel implementation in C++ through the OpenMP environment. The input bit string is divided into different chunks of similar size, the number of such chunks being equal to the number of threads executing in the parallel environment. The chunks are encoded in parallel, and are then appended sequentially to the output bit string.

One consequence of executing encoding algorithms in a parallel environment is that the formation of chunks may split the runs present. For example, if there are 2 threads and the bit string is ‘10000001’, the threads divide the string at the midpoint, thereby splitting the run of 0s. This may result in a different output. However this does not corrupt the encoded string, and can be decoded to give the original run, as decoding involves observing the value of the initial bits in the encoded run.

This is not the case when running the decoding algorithms in a parallel environment The decoding algorithms would require prior knowledge of the size of the runs before dividing the data into chunks, which is very hard and inefficient. Hence, it is preferable not to execute the decoding algorithms in a parallel environment. Algorithm 2 is used to decode both the parallel and serial encoding outputs.

Algorithm 3: Parallel VaFLE Encoding

```
Input: Original file, and threshold length Threshold, number of threads numT
Output: Compressed file
function VaFLE_encoding(file, Threshold, numT )
    s1 ← * file
    s2 ← * compressedFile
    writeFile(s2, Threshold)
    size ← sizeof(file)
    cmprs[numT ] ← NULL
    #pragma omp parallel num_threads(numT )
        t ← omp_get_thread_num()
        beg ← t * size/numT
        end ← (t + 1) * size/numT
        if end ≥ size then
            lt ← sz
        chunk ← bits(s1_beg , lt − beg)
        chunksize ← sizeof(chunk)
        iRun ← 0
        while iRun < chunksize do
            count ← length of current run
            if count ≤ Threshold then
                append(cmprs_t ,bits(chunk_iRun , count))
            else
                value ← bits(chunk_iRun , Threshold)
                flag ← NULL
                flagSize ← log(count − (Threshold - 1))
                append(flag,intToBits(2^(flagSize×chunk iRun) − 1, flagSize))
                append(flag, !(bit(chunk_iRun )))
                length ← NULL
                lSize ← flagSize
                offset ← count − (Threshold − 1) - 2^lSize
                append(length, intToBits(offset, lSize))
                append(cmprs_t , value + f lag + length)
            iRun ← iRun + count
    for j ∈ 1, ...numT do
        writeFile(s2, cmprs_j )
    return compressedFile
```

4 Experimental Methodology

The performance of the parallelized VaFLE algorithm was evaluated and compared with a basic implementation of the RLE algorithm using various lossy and lossless formats and sizes of the standard Lenna image (shown in Table 1). The hardware configuration consisted of a 56-core server grade Intel Xeon CPU with 125 Gigabytes of RAM. The Relative Speedup (RS) and Relative Compression Ratios (RCR) were calculated using formulae (7) and (8) respectively. For JPEG color images, multiple 512×512 samples of varying quality were considered. The file sizes range from 5 KB to 30 KB. The threshold value τ was set to 2, in order to ignore the encoding of lone and pair bits. Dividing the original file size by the compressed file size gives the compression ratio of an algorithm for a given input. We also compare VaFLE's compression ratio against RLE, LZW [10] and Huffman [5] encoding, three well known algorithms.

Table 1. Image formats used for the performance evaluation of VaFLE

Format	Scheme (Color and dimension in pixels)
PNG	Gray (512 × 512), Color (480 × 480, 512 × 512)
BMP	B/W (512 × 512), Gray (256 × 256, 512 × 512), Color (512 × 512)
JPG	Gray (128 × 128, 256 × 256), Color (512 × 512)
TIF	Gray (64 × 64, 256 × 256, 512 × 512, 1028 × 1028), Color (512 × 512)

5 Results and Analysis

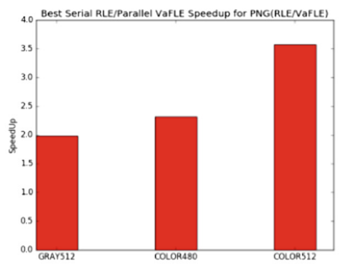
Figure 1 shows the difference in relative execution times achieved while comparing the serial RLE against the parallel VaFLE scheme using Eq. (7). In terms of latency, parallel VaFLE shows a speedup in most of the image formats over serial VaFLE except in JPGs (which are lossy by design, Fig. 1(d). The lossless formats such as PNG (Fig. 1(a)) and BMP (Fig. 1(b)) take the least time to encode. Each thread gets to work on a small portion of the data and quickly completes their encoding task. Parallel VaFLE on most image formats gives a speedup roughly proportional to the image size due to reduced overhead during communication between threads.

$$RS = \frac{\text{Serial RLE Execution Time}}{\text{Parallel VaFLE Execution Time}} \tag{7}$$

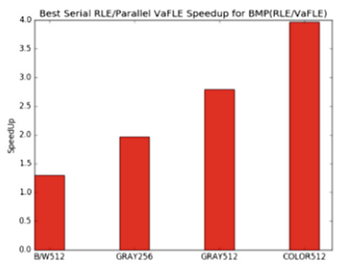
$$RCR = \frac{\text{VaFLE compression ratio}}{\text{RLE compression ratio}} \tag{8}$$

It can be inferred from Fig. 2 that the VaFLE algorithm gives performs similarly in serial and parallel environments, both being relatively better than RLE, although there are minute differences in the compression ratio upon parallelizing VaFLE. Although the same data gets compressed in a manner similar to the serial algorithm, splitting up image data will result in the breaking of long sequences of consecutive bits.

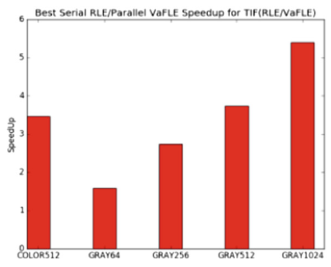
Figure 3 shows the compression ratio of VaFLE, LZW, Huffman and RLE encoding. LZW is a very effective, general purpose algorithm and achieves more the 2X compression ratio for most image formats (30X in the case of the black and white image). While Huffman encoding surpasses the compression ratio performance of VaFLE, it must be noted that Huffman coding spends some time mapping image intensity values to an 8 bit representation. VaFLE does not have such a computational overhead and will have a better runtime compared to Huffman encoding. VaFLE does perform better (10X compression ratio) than Huffman encoding (5X compression ratio) on the black and white image which shows the effectiveness of a Run Length Encoding based scheme when the input contains a large number of similar bits in a sequence.



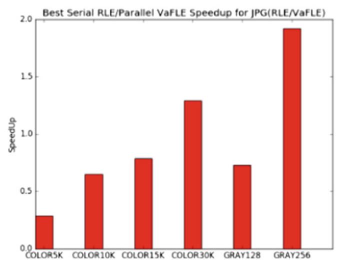
(a) PNG



(b) BMP



(c) TIF



(d) JPG

Fig. 1. Speedup achieved by parallel VaFLE against serial RLE in various image formats.

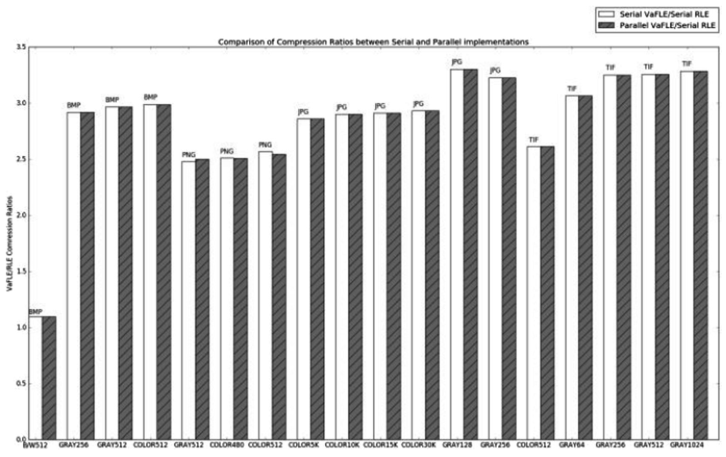


Fig. 2. Comparison of serial and parallel compression ratios (VaFLE/RLE)

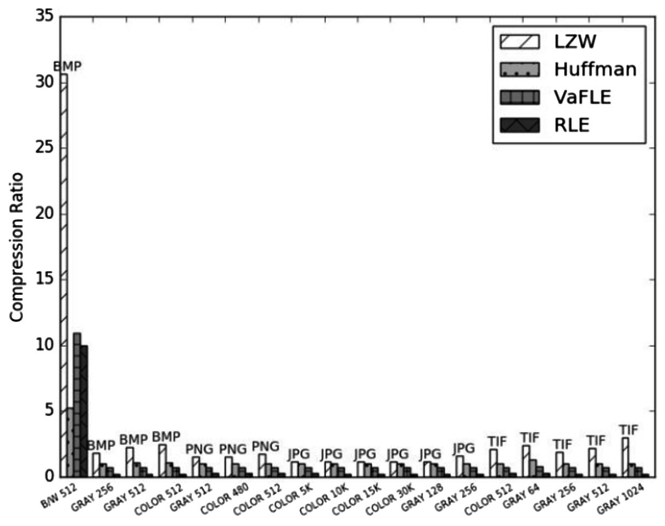


Fig. 3. Comparison of LZW, Huffman and VaFLE

Performance is based on the type of image format and whether the image is black and white, grayscale or colored. JPG attain the highest compression ratios but they are lossy formats, which defeats the purpose of using a lossless algorithm in order to preserve raw image data. Lossless formats, such as PNG, TIF, and BMP, are more favourable media to demonstrate high compression ratios while still retaining the original pixel information.

One limitation seen in the current implementation of the proposed algorithm is that the threshold, τ, is set as an arbitrary constant, which would not give an optimal compression for all cases. While there is no precomputation done before the file is being compressed, it could be done in order to find an optimal τ for that file. However, setting the value of τ beforehand would still be useful in certain cases, such as online compression of data streams, where precomputation should be as minimal as possible.

6 Conclusion and Future Work

In this work, we presented a novel lossless data encoding algorithm called VaFLE. On its own, VaFLE offers much better compression ratios with a negligible increase in runtime compared to the naive Run Length Encoding scheme. A parallelized implementation of VaFLE in OpenMP was tested against similar images with various formats such as BMP, JPG, PNG and TIF. VaFLE performed exceptionally well on Black and White images. Our algorithm guarantees better compression rates of upto 3X greater than standard RLE. Parallelized VaFLE attains a speedup as high as 5X in grayscale and 4X in color images compared to the naive RLE approach. Our implementation of this algorithm can be particularly useful for online compression of data streams. Being a Run Length Encoding based algorithm, VaFLE does not have better compression rates compared to other well known encoding schemes such as LZW and Huffman encoding. However, VaFLE is faster compared to Huffman encoding as there are no pre-computation steps involved.

In the future, we plan to implement an optimized version of this algorithm, in which the optimal value of the threshold length τ can be calculated for a given file before compression takes place, so as to improve the compression rate. Also, since it performs well with binary images or with images where there is a low number of varying intensity values, it is possible for grayscale image intensities to be mapped to a lower number of intensity values and appended as a header so that the mapped image has a high compression ratio but the entire grayscale image can still be reconstructed at the receiver end using the header information. Furthermore, we plan to implement VaFLE using CUDA [7] and further improve the runtime of the algorithm using a large number of GPU threads.

Acknowledgements. We sincerely thank Dr. Sidney Rosario, Department of Information Technology, NITK Surathkal, who helped give us the opportunity to work on this project, and providing valuable inputs during the development of the algorithm itself. We also thank Dr. Basavaraj Talwar, Department of Computer Science, NITK Surathkal, and Mr. Swaroop Ranganath for providing valuable inputs on how the paper could be improved.

Finally, we thank the Department of Information Technology for providing to us the necessary computer facilities and infrastructure needed to implement our project.

References

1. Amin, A., Qureshi, H.A., Junaid, M., Habib, M.Y., Anjum, W.: Modified run length encoding scheme with introduction of bit stuffing for efficient data compression. In: 2011 International Conference for Internet Technology and Secured Transactions, pp. 668–672, December 2011
2. Arif, M., Anand, R.S.: Run length encoding for speech data compression. In: 2012 IEEE International Conference on Computational Intelligence and Computing Research, pp. 1–5, December 2012. https://doi.org/10.1109/ICCIC.2012.6510185
3. Chakraborty, D., Banerjee, S.: Efficient lossless colour image compression using run length encoding and special character replacement (2011)
4. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
5. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40** (9), 1098–1101 (1952). https://doi.org/10.1109/JRPROC.1952.273898
6. Khan, S., Khan, T., Naem, M., Ahmad, N.: Run-length encoding based lossless compressed image steganography. SURJ **47**, 541–544 (2015)
7. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500
8. Trein, J., Schwarzbacher, A.T., Hoppe, B., Noff, K.: A hardware implementation of a run length encoding compression algorithm with parallel inputs. In: IET Irish Signals and Systems Conference (ISSC 2008). pp. 337–342, June 2008. https://doi.org/10.1049/cp:20080685

9. Vijayvargiya, G., Silakari, S., Pandey, R.: A survey: various techniques of image compression. CoRR abs/1311.6877 (2013). http://arxiv.org/abs/1311.6877
10. Welch, T.A.: A technique for high-performance data compression. Computer **17**(6), 8–19 (1984). https://doi.org/10.1109/MC.1984.1659158