The Journal of Supercomputing (2022) 78:2681–2708
https://doi.org/10.1007/s11227-021-03971-6

An efficient parallel entropy coding method for JPEG compression based on GPU

Fushun Zhu[1] · Hua Yan[1]

Accepted: 22 June 2021 / Published online: 5 July 2021

Abstract
The fast JPEG image compression algorithm is a requisite in many applications such as high-speed video measurement systems and digital cinema. Many existing methods have implemented the JPEG compression in parallel based on GPU except for entropy coding, which is a variable-length coding method and seems like a better fit for sequential implementation. However, entropy coding is an essential part of the JPEG compression system and typically takes up a large proportion of the time when implemented on the CPU. To tackle this problem, we propose an efficient parallel entropy coding (EPEnt) method for parallel JPEG compressing. The proposed method conducts entropy coding in three parallel steps: coding, shifting, and stuffing. Specifically, according to the different characteristics of image components, we devise thread-based and warp-based functions in the coding stage to further improve the efficiency under guaranteeing image quality, respectively. We apply the proposed method to the parallel JPEG compression system and evaluate the performance based on compute unified device architecture (CUDA). The experimental results demonstrate that compared with sequential implementation, the maximum speedup ratio of entropy coding can reach 39 times without affecting compressed images quality. Meanwhile, the whole JPEG compression process efficiency increases by at least 28% compared with state-of-the-art parallel methods in terms of speedup ratio.

Keywords Real-time systems · JPEG · Entropy coding · Parallel algorithm · CUDA

✉ Hua Yan
yanhua@scu.edu.cn

[1] College of Electronics and Information Engineering, Sichuan University, Chengdu 610065, Sichuan, China

1 Introduction

With the continuous development of high-performance computing (HPC) applications, real-time image compression has gained considerable attention, such as high-speed video measurement systems and digital cinema [1–3]. On many occasions, numerous large-size images are obtained from these systems rapidly, and these source images are required to be quickly stored for further study in real-time. However, it is difficult to directly transmit and store them, characterized by large amounts of data and high redundancy. Therefore, the JPEG compression algorithm [4] plays an irreplaceable role in these HPC applications owing to its high compression efficiency. Unfortunately, the sequential implementation, especially for the large-size images, is very slow and cannot guarantee the real-time performance of these HPC applications [5]. We hope to explore a method that can increase the image compression speed without affecting the compressed image quality.

GPU can launch thousands of threads simultaneously, and it is eminently suitable for parallel computing and significantly speeds up the process than sequential algorithms. Many HPC applications have been completed a parallel job to improve system efficiencies such as image representation and recognition [6, 7], image reconstruction [8], and super image resolution [9]. The parallel computation based on GPU has become more extensive, especially when CUDA and Open Computing Language (OpenCL) technologies are becoming mature. Mainly, CUDA is the most popular parallel computational framework owing to its convenient and efficient operating platform. Therefore, it is pretty necessary to implement a JPEG parallel algorithm based on CUDA to meet the requirements of HPC applications.

As shown in Fig. 1, there are six steps to realize the JPEG compression algorithm: color conversion, down-sampling, forward 2D Discrete Cosine Transform (DCT), quantization, zig-zag scan, and entropy coding. Forward 2D DCT and entropy coding are the most important and time-consuming parts. They can remove spatial redundancy and structural redundancy of image data, respectively.

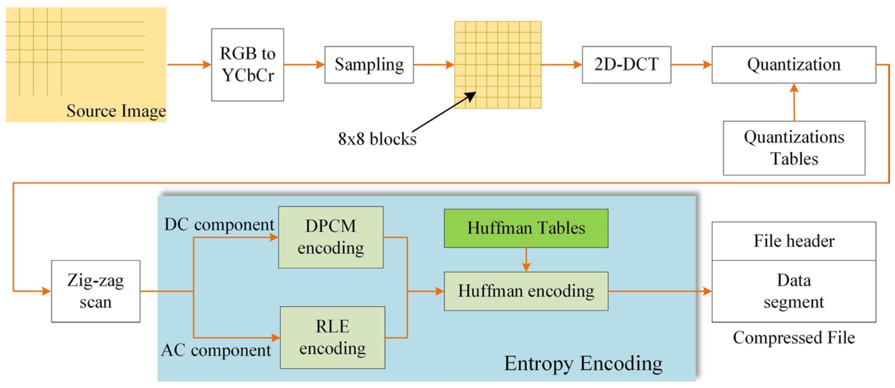


Fig. 1 The diagram of JPEG image compression algorithm

Many researchers have concentrated on parallelizing the JPEG algorithm based on GPU. Especially in forward 2D DCT, promising results have been achieved in acceleration and efficiency [10–17]. Tokdemir et al. [17] proposed a DCT parallel algorithm to accelerate data processing and obtained satisfactory results. Liu et al. [18] presented a parallel DCT algorithm for the JPEG algorithm. The parallel DCT algorithm has achieved up to 20 $\times$ speedup compared with the sequential DCT algorithm in their experiment. Alqudami et al. [10] developed an optimized parallel implementation of the forward DCT algorithm based on OpenCL. New parallel data transform methods were proposed to replace the parallel DCT algorithm for better efficiency and image compression quality [15, 16]. Parallel DCT-like algorithms [12] were proposed to substitute for the parallel DCT algorithm to achieve a faster speed. However, none of the above approaches has completely realized the parallel entropy coding, making it still a bottleneck in parallel JPEG image compression.

Entropy coding plays an indispensable role in JPEG image compression, aiming to remove the structural redundancy of image data. In general, when the JPEG algorithm is executed based on CPU, the complex entropy coding is inefficient, and it will take a lot of storage and computing resources. Therefore, it is essential to parallelize entropy coding to improve the real-time performance of JPEG compression. However, there are still challenges due to the following reasons: (1) Based on 8×8 block, entropy coding is a complicated procedure, and many classic algorithms are adopted to generate the coding results of each block. Different from the sequential implementation, an optimal parallel entropy coding strategy is required to obtain the coding results quickly. (2) The bit length of each 8×8 block coding results is variable and always ranges from several to hundreds of bits in length. Therefore, even if the entropy coding is performed in parallel, it is still not clear where the result of each 8×8 block should be written into the output image data. Furthermore, the connection part of neighboring 8×8 block coding results may be written to the same byte simultaneously, leading to the possibility of access collision and write fault.

Due to the challenges mentioned above, some researchers believe that entropy coding should be processed on the CPU when the JPEG parallel algorithm is employed [18, 21, 22]. However, many other researchers have focused on solving this problem from different aspects [16, 18, 21, 23, 34, 35]. Shan et al. [15, 16, 28] proposed a parallel entropy coding method for JPEG image compression based on CUDA, but they did not solve how to generate the final image data in parallel by utilizing all the 8×8 block coding results. Instead of color images, only gray images were adopted to verify the performance, limiting the application of their method. In [20], Atomic operations were fully used for variable-length encoding, limiting GPU's performance. Rahmani et al. [23, 24] introduced a parallel prefix sum algorithm [25–27] to realize parallel Huffman encoding of strings, and the encoding speed was improved significantly compared with the sequential implementation. In this algorithm, each character's coding length must be obtained quickly, and a vast amount of temporary memory is consumed. Before calculating each 8×8 block's coding bit length, all these blocks should be encoded firstly, which is a complex process. Therefore, it is not easy to directly employ the algorithm to JPEG entropy coding. Tian et al. [34] developed an efficient parallel codebook construction and a novel reduction-based encoding scheme to implement a multi-thread Huffman encoder. Extensive experiments on six real-world datasets were conducted to evaluate the superior performance of their method. In [35], a new data structure called gap array was presented to accelerate the Huffman decoding. Although these algorithms solved the problem of realizing the parallel encoding with variable-length codes in some way, the challenges in parallel entropy coding mentioned above are not adequately addressed. Recently, CUDA has provided a powerful image parallel processing library named NVIDIA Performance Primitives (NPP) [29]. The latest version (NPP v10.2) already provides a functional interface for parallel entropy encoding without open-source codes. Compared with sequential implementation, the performance is much improved. However, parallel entropy coding is still a bottleneck in parallel JPEG image compression.

In this paper, we also focus on overcoming the above challenges and developing an efficient parallel method for entropy coding named EPEnt. The proposed method mainly contains three steps: coding, shifting, and stuffing. In the coding phase, all the 8×8 blocks of an image are encoded to generate their corresponding coding results. Moreover, according to the different characteristics of image components, warp-based and thread-based strategies are designed to improve the coding stage's efficiency. In the shifting phase, appropriate bit shift operations for each 8×8 block coding result ensure that the neighboring block bitstreams can be seamlessly stitched together when forming the final image data. Finally, All the shifted results of 8×8 blocks are stitched together to form the final image data in the stuffing phase. The proposed EPEnt is evaluated with multiple sets of experiments and achieves promising speedup and image quality results. Meanwhile, the performance of the JPEG image compression also improves significantly when our proposed method is employed.

Overall, the contributions of our work are summarized as follows:

- We propose an efficient parallel entropy coding method named EPEnt to improve the performance JPEG image compression. The proposed method mainly consists of three phases: coding, shifting, and stuffing.
- According to the different characteristics of image components, we, respectively, design warp-based and thread-based strategies to complete the coding stage, which can further improve the entropy coding efficiency.
- We apply our proposed method to the parallel JPEG compression algorithm. Experimental results demonstrate that our method can achieve competitive results in terms of speed ratio and image quality.

The remainder of the paper is organized as below: Section 2 describes the entropy coding and CUDA background. Section 3 introduces the proposed methods. The experimental results and performance evaluation are presented in Sect. 4. Finally, we draw our conclusions and outline future work in Sect. 5.

2 Background

2.1 JPEG entropy coding

2D DCT transforms an image from the spatial domain to the frequency domain to remove spatial redundancy. Entropy coding mainly adjusts the code length of the transformed data, which can further improve the compression ratio and generate the final image data. The sequential entropy coding algorithm process of each 8×8 block mainly contains three stages: differential pulse code modulation (DPCM), run-length encoding (RLE) coding, and Huffman coding, and we will describe the details in the following procedure.

The two-dimensional matrix is transformed into a 1×64 one-dimensional array after the 8×8 block is scanned in a zig-zag way. The one-dimensional array is composed of two parts. The first value in the array is direct current (DC) coefficient, while the remaining values are alternating current (AC) coefficients. We perform DPCM and RLE coding on these two coefficients separately to form the temporary codes that require to be continuously processed by the Huffman coder.

The DC coefficient contains the primary information of an 8×8 block, and the value is generally immense. However, the difference between two adjacent 8×8 blocks is small and ranges in a small region. Therefore, it is fit for DPCM coding. The difference value D_k of each 8×8 block is consequently defined as:

$$D_k = \begin{cases} DC_k - DC_{k-1}, & \text{if } k = 2, 3, \ldots, N \\ DC_k, & \text{if } k = 1 \end{cases} \tag{1}$$

where N denotes the total number of 8×8 blocks in an image component, and k is the number of an 8×8 block.

Then, D_k can be described by two symbols, as shown in Fig. 2a. S represents the size of $D_{k,}$ and A is the amplitude of D_k. The code of S is obtained through querying the DC Huffman table, and the code of A is obtained by a variable-length integer (VLI), which can be described as follows:

$$D_k = \begin{cases} D_k, & \text{if } D_k \geq 0 \\ 2^{\text{size}} - |D_k| - 1, & \text{if } D_k < 0 \end{cases} \tag{2}$$

The code of A is put at the end of the code of S to constitute the Huffman code of DC coefficient.

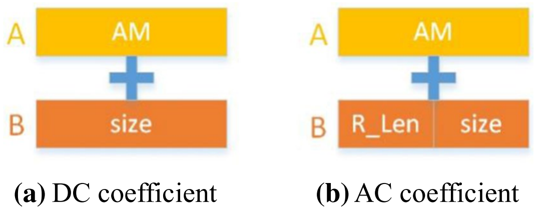


Fig. 2 The Huffman code of DC/AC coefficients

The run-length and the size of non-zero AC coefficients can also be described by two symbols, as shown in Fig. 2b. *S* contains run-length and the size of non-zero AC coefficients code. *A* is the amplitude of AC coefficients. The Huffman code of *S* can be obtained through querying the AC Huffman table, while the code of A is obtained by employing VLI.

Consequently, the entropy coding results of each 8 × 8 block are obtained by encoding the DC coefficient and AC coefficients using Huffman coding utilizing Huffman algorithms. All the entropy coding results of 8 × 8 blocks are stitched together in sequential to form the final image data.

2.2 CUDA programming model

CUDA is a GPU-based parallel computing architecture. Users can easily program with CPU + GPU based on CUDA. CPU acts as the host responsible for the overall logic control, task scheduling, and resource management in CUDA architecture. GPU is controlled by the CPU and acts as the device to handle highly parallel computing tasks [29].

When processing tasks in parallel based on GPU, we use a kernel function to represent the same subtasks. It will be called in parallel by a large number of CUDA threads. As shown in Fig. 3, the threads are equally divided into a certain amount of blocks, and the threads in a block are processed in groups based

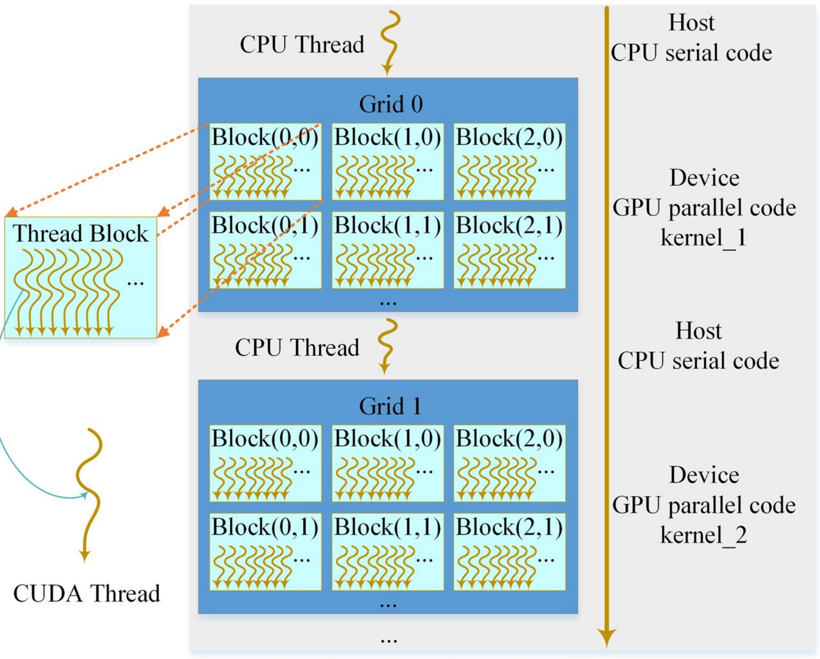


Fig. 3 The CUDA programming model

on warps. At present, each warp contains 32 threads, which execute the same instructions but process different data. Therefore, the thread-block size is generally an integral multiple of 32. These blocks are organized into a grid that can fully utilize the hardware resources [30]. A complete CUDA program consists of multiple kernel functions and CPU sequential code.

3 CUDA-based parallel entropy coding (EPEnt)

3.1 Algorithmic overview

The procedure of JPEG compression is performed based on 8×8 block. Aside from entropy coding, each block in other stages can be processed independently and is desirable for parallel processing. The main reasons are chiefly as follows: (1) There is a lack of efficient strategy to encode all 8×8 blocks in parallel. (2) Owing to the variable-length coding result of each 8×8 block, it is not clear where the threads should write the corresponding coding results into the final image data. Furthermore, the write operation in the same memory may be performed by two adjacent threads during parallel implementation, which could lead to an access violation. Therefore, an excellent parallel strategy is required to put all the 8×8 blocks' coding results together to form the final image data.

To address the above issues, we propose an efficient parallel entropy coding method called EPEnt for JPEG image compression, which is shown in Fig. 4. The fundamental idea of EPEnt is that the whole entropy coding process can be conducted through three parallel steps: coding, shifting, and stuffing, with details as follows:

(1) *Coding* All 8×8 blocks in an image are encoded in parallel to form their respective bitstream. Because of an apparent contrast between the task loads of Y

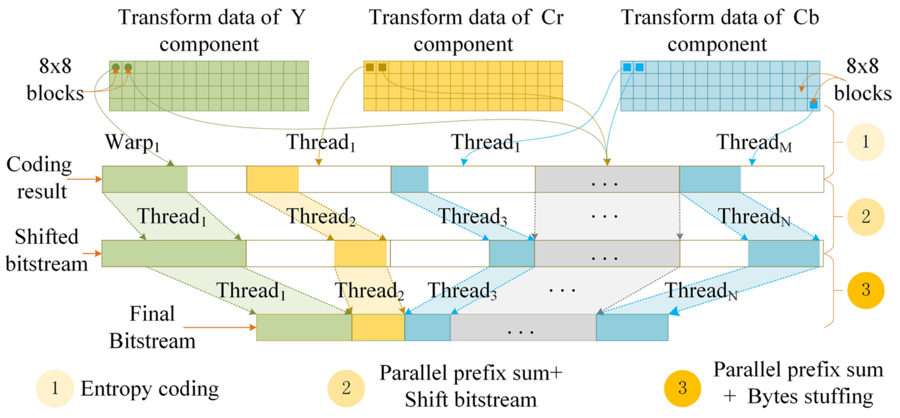


Fig. 4 Illustration of the proposed method for parallel entropy coding

and CbCr components, the thread-based and warp-based kernel functions are, respectively, designed for a more efficient coding process.

(2) *Shifting* According to the result of the prefix sum algorithm (PSA), appropriate shift operations for each 8×8 block's coding result will ensure that the adjacent block's bitstream can be seamlessly stitched together when forming the final image data.

(3) *Stuffing* As step 3 described in Fig. 4, all the 8×8 blocks' shifted results are stitched together to form the final image data based on start position, which is also calculated via PSA.

It is noteworthy that two key parameters (i.e., offset and start position) are required to be calculated quickly for the second and third phase of EPEnt. To tackle this problem, the PSA plays a crucial role, and we modify it slightly to implement our task. Moreover, the detailed implementation will be given in Sect. 3. 2.

3.2 Pipelined execution scheme of EPEnt

3.2.1 Encoding transformed data

As previously stated, a JPEG image contains three components (luminance component Y and chrominance components CbCr) which are significantly different. To be specific, Y contains the primary information of an image, and the change of pixel value evidently affects people's perception of the image. On the contrary, the chrominance components include less image information, and people are less sensitive to it than Y.

Algorithm 1: Parallel_Y_component_coding (Y_block, TempJPEGdata, Huffman_Tab)

```
//input: Y_block, an 8x8 block that needs to be encoded
//input: Huffman_Tab, the Huffman table stored in constant memory
//output: TempJPEGdata, temporary storage for the parallel coding results
*allocate shared memory Temp[32] for this 8×8 block
*retrieve thread ID(ix) in the warp
In_first = Y_block[2 × ix]                    // 0≤ ix < 32
In_second = Y_block[2 × ix + 1]               // 0≤ ix < 32
If ix = 0 then
    In_first ← DPCM coding on In_first
End If
// perform RLE coding on In_first, In_second
// zeros counting by using warp vote function "__ballot()"
nonzero_first_flag = 1 | _ballot(In_first)
nonzero_second_flag = _ballot(In_second)
zeros_before_first, zeros_before_second ← zeroscount (nonzero_first_flag, nonzero_second_flag)
//perform the Huffman coding to generate bitstream
first_bits ← Huffman_coding (zeros_before_first, In_first, Huffman_Tab)
second_bits ← Huffman_coding (zeros_before_second, In_second, Huffman_Tab)
bits_length ← calculate_bits_length(first_bits, second_bits)
Temp[ix] = bits_length
__syncthreads()
// utilize prefix sum to calculate the offset of the thread's encoding bitstream in TempJPEGdata
For stride = 1; stride <= ix; stride = stride×2
    __syncthreads()
    len = Temp[ix-stride]
    __syncthreads()
    Temp[ix] += len
End For
bits_offset = Temp[ix] - bits_length    //calculate the start position for corresponding coefficient in the 8×8 block
  If bits_length != 0 then              //start writing the Huffman code to the temporary storage
      *write the first_bits and second_bits to the TempJPEGdata according to the bits_offset
  End If
```

Based on the above background, we first attempt to analyze the differences between the coding task loads of both components. We observe that the number of non-zero coefficients directly determines the task loads of an 8×8 block because they are the main target which needs to be converted to the bitstream. Therefore, we choose several typical standard images and analyze their task loads. Figure 5 presents the functional relationship between the number of 8×8 blocks and the number of non-zero coefficients in each image component. From it, we can summarize that each 8×8 block in luminance component Y contains a lot of non-zero coefficients, which will lead to a heavy workload if only a single thread is performed. Nevertheless, in chrominance components CbCr, there are only a few non-zero coefficients in each 8×8 block, which can be efficiently encoded even by just one thread.

In our proposed EPEnt, two different coding strategies are, respectively, devised for Y and CbCr components. The main difference between both strategies is that

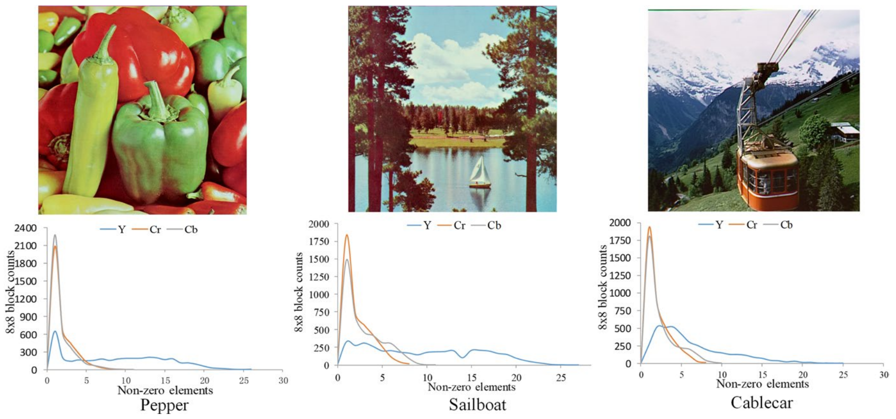


Fig. 5 The distribution of the number of non-zero coefficients of 8 × 8 block in each image component

each 8 × 8 block of the Y component is encoded based on the CUDA warp, while only one thread is required to complete the same task CbCr components.

Algorithm 1 presents the details of a warp-based coding strategy, where 32 threads in a warp are responsible for encoding two adjacent coefficients in an 8 × 8 block. First, we can observe that the first thread performs a DPCM algorithm on the DC coefficient. Then all 32 threads simultaneously carry out RLE algorithm on AC coefficients. Notably, the main purpose of RLE coding is to get the run-length by counting the zero coefficients before a non-zero coefficient.Instead of counting zeros directly, we introduce the warp vote function *__ballot()* so that 32 threads can work in concert to accomplish this task. Afterward, each thread obtains the bitstream and its corresponding bit length *bits_length* by employing the Huffman algorithm on the RLE coding results.

At this time, threads cannot determine where their obtained bitstream should be written in the final image data owing to the variable length codes. In our EPEnt, a 64B of global memory (i.e., *TempJPEGdata*) is allocated for each 8 × 8 block to store the bitstream temporarily, and the memory size can be adjusted if needed. In general, the extra storage memory required is very small, which is only equivalent to the storage space of source image. To calculate the start position where bitstream is written into *TempJPEGdata*, all threads write their corresponding (i.e., *bits_length*) to allocated shared memory *Temp*. The prefix sum is adopted to calculate the start position where their bitstream should be written into *TempJPEGdata*. And we use the *__syncthreads()* function to avoid the parallel write-conflict error.

Algorithm 2: Parallel_ CrCb _component_coding (CrCb_block, TempJPEGdata, Huffman_Tab)

```
//input: CrCb_block, the 8x8 block that needs to be encoded
//input: Huffman_Tab, the Huffman table stored in constant memory
//output: TempJPEGdata, temporary storage for the parallel coding results
//encode the DC coefficient
temp ← DPCM coding on CrCb_block [0]
bitsbuff ← Huffman_coding (temp, Huffman_Tab)
bits_length = calculate_bits_length(bitsbuff)
zeros_before_temp = 0
*write the bitsbuff to the TempJPEGdata  according to the bits_offset
bits_offset = bits_length
//encode the AC coefficients
For i = 1:64
    temp = CrCb_block [i]
    If temp != 0 then
        bitsbuff ← Huffman_coding (zeros_before_temp, temp, Huffman_Tab)
        bits_length ← calculate_bits_length(bitsbuff)
        *write the bitsbuff to the TempJPEGdata according to the bits_offset
        bits_offset += bits_length
        zeros_before_temp = 0
   Else
        zeros_before_temp = zeros_before_temp + 1
   End If
End For
```

As mentioned above, we utilize a thread-based strategy to encode an 8×8 block of CbCr components, as shown in Algorithm 2. Similar to CPU-based methods, and the run length is obtained by just counting zero coefficients. When obtaining the bitstream after RLE and Huffman algorithms, threads write the bitstream into the corresponding *TempJPEGdata* in a sequential way.

Therefore, the coding results of all 8×8 blocks are written into their corresponding *TempJPEGdata* after the coding phase. And the bit length of the coding results can also be obtained. For example, in Fig. 6, the bit length l of the k_{th} 8×8 block is 17; the bit length l of the $k+1_{th}$ 8×8 block is 4; the bit length l of the $k+2_{th}$ 8×8 block is 11. The final output result of the coding phase is also described as the coding result in Fig. 4.

3.2.2 Implementation of shifting

As shown in Fig. 6, all the 8×8 block coding results are written into their corresponding *TempJPEGdata* beginning from the first-bit position for higher parallel efficiency. Unfortunately, the bit length of most 8×8 block coding results is not an

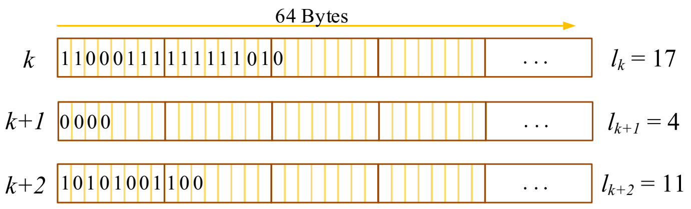


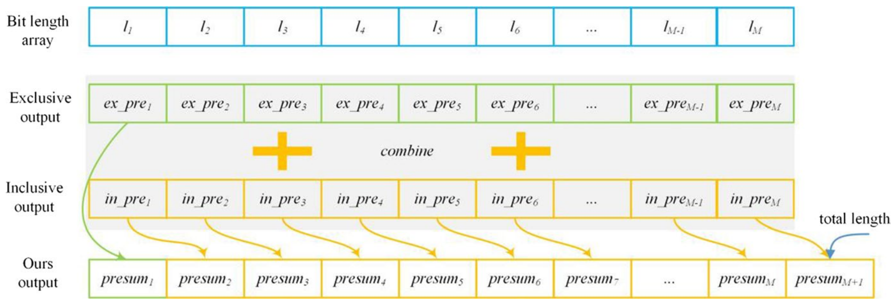
Fig. 6 The visualization result of coding phase. We allocate a 64B of temporary storage *TempJPEGdata* for each 8×8 block. And the coding results are written into their corresponding *TempJPEGdata* from left to right

integral multiple of byte, which means that the coding results of adjacent blocks cannot be stitched together continuously in the current state, resulting in incorrect image data. Therefore, we need to calculate the bit offset for each 8×8 block coding result and carry out appropriate shift operations to ensure that the neighboring bit-streams can be stitched without bit separation.

By obtaining the start position of each 8×8 block coding results in the final image data, the corresponding bit offset is easily computed. First, the start position of each 8×8 block coding result is calculated by accumulating all bit lengths, which are stored precedently in the final output stream. Then the bit offset of each 8×8 block coding result can be calculated through their start position. Inspired by the excellent parallel performance of PSA, we introduce and slightly modify it to obtain the start position and meet the demands of our proposed EPEnt. Next, we will provide the detailed implementation process.

As shown in Fig. 7, when the PSA is operated on a bit length array, it could generate two output arrays by performing inclusive and exclusive scans, respectively. Inclusive scan produces a new array where each element in_pre_k is the sum of all elements from l_1 to l_k,, which can be described as follows:

$$in\_pre_k = \sum_{i=1}^{k} l_i \tag{3}$$



Bit length array
l_1 l_2 l_3 l_4 l_5 l_6 ... l_{M-1} l_M
Exclusive output
ex_pre_1 ex_pre_2 ex_pre_3 ex_pre_4 ex_pre_5 ex_pre_6 ... ex_pre_{M-1} ex_pre_M
combine
Inclusive output
in_pre_1 in_pre_2 in_pre_3 in_pre_4 in_pre_5 in_pre_6 ... in_pre_{M-1} in_pre_M
total length
Ours output
$presum_1$ $presum_2$ $presum_3$ $presum_4$ $presum_5$ $presum_6$ $presum_7$ ... $presum_M$ $presum_{M+1}$

Fig. 7 Parallel prefix sum scan on the bit length array

Also, exclusive scan is helpful to our proposed method. Each element *ex_pre_k* computed by exclusive scan contains the sum of all previous elements ($l_1 \rightarrow l_{k-1}$), but not l_k itself, and the process is defined as:

$$ex\_pre_k = \begin{cases} 0, & \text{if } k = 1 \\ \sum_{i=1}^{k-1} l_i\,, & \text{if k} \geq 2 \end{cases} \tag{4}$$

The start positions and offsets of 8 × 8 block coding results are calculated based on the exclusive scan. Furthermore, the total bit length of the entire image data is also a key parameter for generating JPEG files, which can be obtained via inclusive scan. Therefore, when calculating these parameters by PSA, we set the first value of the output array to 0, and then arrange the result of the inclusive scan into it, as shown in Fig. 7. In this way, these required parameters can be calculated merely from an output array.

Algorithm 3 presents how to shift each 8 × 8 block coding results by utilizing the bit offset s. Firstly, the bit offset *s* of each 8 × 8 block coding result is calculated according to the corresponding start position *presum.* Afterward, we perform shift operation on all 8 × 8 block coding results in byte according to their related offset *s.* Finally, to save the memory source and not affect the shift result, we start shifting from the last byte of the coding results. The newly generated byte is written starting from the last byte of each temporary storage *TempJPEGdata.* Therefore,we will implement the phase without any extra storage.

Algorithm 3: Parallel_shifting_coding_results(TempJPEGdata, presum, bit_length)

```
//input: TempJPEGdata, temporary storage for the parallel 8×8 block coding results
//input: presum, the total bit length before the 8×8 block coding results
//input: bit_length, the bit length of the 8×8 block coding results
//output: TempJPEGdata, temporary storage for the parallel shifting results
//calculate the offset s
$s = (presum \bmod 8)$
$byte\_l = (l + 7) / 8$        //calculate the byte length of the 8×8 block coding results
$Tempsize = 64$              //the byte length of the temporary storage TempJPEGdata
//shift the coding results and find 0xff byte
For $i$ = byte_l; $i > 0$ ; $i$--
    $new\_byte$ = (TempJPEGdata[$i$] >> $s$) +(TempJPEGdata[$i$-1] << (8-$s$))
    If $new\_byte$ = 0xff then
        TempJPEGdata[$Tempsize$ --] = $0x00$
        $bit\_length = bit\_length + 8$
    End If
    TempJPEGdata[$Tempsize$ --] = $new\_byte$
End For
```

In addition, 0xff byte in JPEG files indicates the identifier that might also be produced in the image data. Based on JPEG standard, 0×00 byte will be added after the 0xff byte when it appears in the image data. In the shifting phase, we can directly judge whether the newly generated bytes are 0xff or not. And the 0×00 will be put behind the 0xff to distinguish between the image data and the identifier. It is noteworthy that after the operation completes, the shifting results can also guarantee the stitching continuously of adjacent block coding results.

Figure 8 provides the bit offset s_k of the k_{th} block shown in Fig. 6, all the 17 bits will be shifted to the right by 2 bits. The bit length will also be changed from 17 to 25 due to the 0xff byte after shifting. Similarly, the $k+1_{th}$ and $k+2_{th}$ 8×8 block coding results will also be shifted according to their respective offset s. We can obtain the visualization result of the shifting phase in Fig. 4, described as the shifted bitstream.

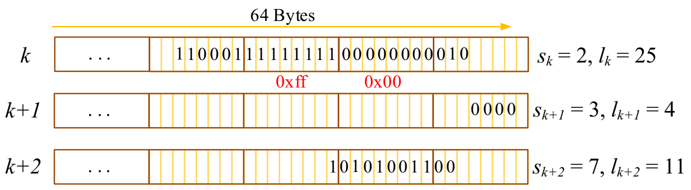


Fig. 8 Shift the 8×8 block coding results and find 0xff bytes

3.2.3 Producing image data by stuffing

The bit length of each 8×8 block coding results changes uncertainly owing to the addition of 0×00 bytes after the shifting phase. Therefore, we conduct the modified PSA again to compute the start position *Sp* for each 8×8 block to decide where to start writing. The total length of the final image data is also obtained during the process. According to the *Sp,* threads can write the 8×8 block shifting results to the corresponding location accurately in the final image data, which is allocated during initialization. However, supposing that adjacent threads write the bitstreams simultaneously, they may need to access the same memory location. Thus, it is indisputable that access violation and write error will occur, which is shown in Fig. 9.

To solve the above problem, all the 8×8 blocks of three image components are divided into three parts (Y_1, Y_2, and Y_3), and each part is regarded as a collection Y_i that can be expressed as follows:

$$Y_{\mathrm{i}} = \begin{cases} 0, 3, ..., \mathrm{int}(\frac{M+2}{3}) \times 3, & \text{if } i = 1 \\ 1, 4, ..., \mathrm{int}(\frac{M+2}{3}) \times 3 + 1, & \text{if } i = 2 \\ 2, 5, ..., \mathrm{int}(\frac{M+2}{3}) \times 3 + 2, & \text{if} i = 3 \end{cases} \tag{5}$$

where M is the total of all 8×8 blocks in an image. During the stuffing process, the parallel stuffing operation is performed three times to generate the final image data. Each time the shifting results of a collection Y_i will be stuffed into the final output stream in parallel, as shown in Fig. 9. Only in this way can we avoid access violation and write error. The main reason is that in the same parallel stuffing operation, neighboring 8×8 blocks are separated by two other 8×8 blocks, which require to be written next time or have already been written into the final output stream. When the AC and DC coefficients are all zeros in an 8×8 block, we will get the shortest shifting results: 4 bits of chrominance component and 6 bits of the luminance component. Therefore, the neighboring 8×8 block shifting results in the same stuffing operation are separated by at least a byte, indicating there is no danger of access violation in our proposed method. Eventually, the final image data will be produced in parallel after the stuffing phase and transmitted to the host for memory storage.

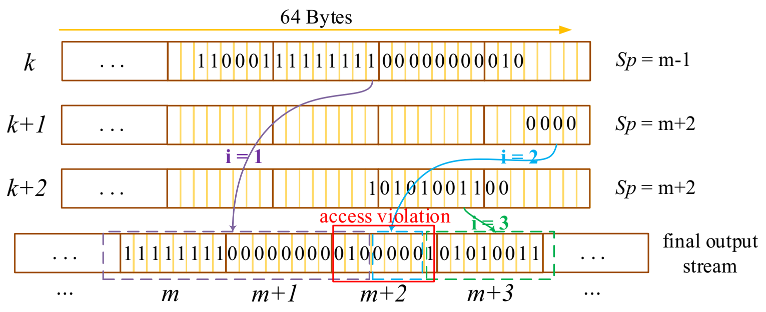


Fig. 9 The shifting results 8×8 block stitching

Besides, the total length of the final image data will also be transmitted to the host for generating a new JPEG file.

3.3 Optimal strategy

GPU is tailor-made for parallel computing, and proper parallel optimization strategies can help our proposed EPEnt run with higher efficiency. The choice of thread-block size has a significant impact on the execution time of kernel functions. The optimal thread-block size depends on the hardware. Therefore, it always requires to specify the thread-block size for better performance. Similarly to previous work [31, 33], the thread-block in our work is set automatically by using the CUDA API-call, *cudaOccupancyMaxPotentialBlockSize()* function, which heuristically calculates a block size that achieves the maximum occupancy. This function is invoked to determine the thread-block sizes of the coding, shifting and stuffing kernels, respectively. Meanwhile, we select images of 2560 × 1600 and encode them on GPU applying our method to compress pictures in 4:4:4 formats.

Figure 10 illustrates kernel execution times versus achieved GPU occupancy as the thread-block size is varied on the GTX1050Ti platform. The CUDA API-calls select the largest possible thread-block size, i.e., 1024 threads. Conversely, we found through trial-and-error that block configurations of 64 threads yielded the minimum execution time when the luminance component coding and stuffing kernels are executed. The other two kernels perform best when the thread-block size is set to 32. However, these differences affect overall JPEG image compression performance by less than 1% on the GTX1050Ti platform. Furthermore, the overhead of this CUDA API-call has no noticeable impact on system performance. We thus employ the CUDA API-call because it selects a close-to-optimal solution without the cost of trial-and-error.

Our proposed method has basically realized the maximum parallel execution of entropy coding. A large number of parallel read and write operations are carried out on the memory, making it easy to cause the bottleneck of the coding system. We should pay attention to balance the usage of different types of memory. In memory optimization, we make full use of constant memory and shared memory. Particularly, cache and broadcast are two main characteristics of constant memory. In our proposed method, we put the Huffman tables in constant memory

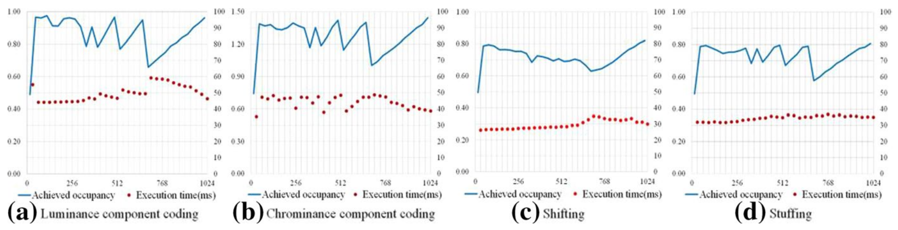


Fig. 10 Entropy coding kernels execution time versus achieved occupancy as the thread-block size is varied

to improve the reading speed. The shared memory is located on the chip, and accessing shared memory is as fast as that of accessing registers. When the 8×8 block bit length is placed in shared memory, the solution of PSA is very fast.

Data transmission between the CPU and GPU has a more considerable influence on the performance of the heterogeneous system [19, 31]. It is always the bottleneck of the parallel algorithm. In order to achieve higher data transmission bandwidth, *cudamallochost()* can be used to allocate page-lock memory on the host. In the meantime, multi-stream technology is utilized to address the transmission delay.

4 Experimental results

4.1 Experimental environment

To evaluate the performance of the proposed method, we conduct several groups of experiments to make a comparison with sequential implementations in image quality and consuming time. In the sequential version, the latest open-source code of libjpeg-turbo's implementation is adopted and modified for comparison purposes. In the JPEG library, the forward 2D DCT has different implementations. We choose the float-type implementation for its accuracy. Also, entropy coding is highly optimized for modern computers. Therefore, the whole process of image compression can be executed quickly on the CPU. The source code for our parallel implementation is developed based on CUDA, and it mainly consists of two parts: the host program and kernel codes. The kernel codes are executed based on GPU and contain the whole process of JPEG image compression.

Table 1 lists the test environment of our proposed method, including hardware and software configuration. Our experiments are carried out on a heterogeneous platform consisting of a multicore CPU and a GPU. Many typical color images with different sizes are employed to support the experiments. And they are compressed to JPEG image with full/down-sampled resolution formats (i.e., 4:4:4, 4:2:2, and 4:2:0). In addition, the standard quantization tables and image quality values (e.g., $Q = 50$) are used to produce the quantization coefficients during the experimental process. Furthermore, we employ the standard Huffman tables to complete the entropy coding stage.

Table 1 Test platform specifications

Hardware platform	Software platform
CPU model: i7-4700 HQ Frequency: 2.4 GHz CPU memory size: 8 GB	Operating system: Windows 10 64 bits Professional Edition
GPU model: GTX 1050Ti GPU memory size: 4 GB Core frequency: 1290 MHz	Software environment: CUDA 10.2

4.2 Visual quality of the compressed images of parallel implementation, and sequential implementation

We cannot improve the compression efficiency without considering the quality of compressed images. Therefore, before evaluating the efficiency of our proposed EPEnt, we firstly compare the image quality achieved by parallel entropy coding and sequential entropy coding respectively. To be specific, two performance metrics, named Peak Signal to Noise Ratio (PSNR) and structural similarity index (SSIM) [32], are introduced to complete the evaluation. The metrics are defined as follows:

$$\mathrm{PSNR} = 10 \times \lg \left[\frac{255HW}{\sum_{i=1}^{W} \sum_{j=1}^{H} \left[x(i,j) - y(i,j) \right]^2} \right] \mathrm{d}B \tag{6}$$

where x and y are the original image and compressed image, respectively, W and H are the dimensions of image array, i and j are the pixel locations.

$$\mathrm{SSIM} = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \times \frac{2\delta_{xy} + C_2}{\delta_x^2 + \delta_y^2 + C_2} \tag{7}$$

where x and y donate the original image and compressed image, respectively, μ_x and μ_y are the mean value of both images, δ_x and δ_y mean the standard deviation of both image, δ_{xy} represents the covariance of both image, C_1 and C_2 are two constants to avoid system errors when the denominator is 0.

The PSNR value range is generally 20–40 dB. The higher the value is, the less the compressed image distort. SSIM, ranging from 0 to 1, is a metric to measure the similarity between two images. Although the calculation is complicated, SSIM is more appropriate to evaluate the image's subjective perception by the human eye than PSNR. Similar to PSNR, the larger the SSIM is, the more similar the compressed image is to the original image.

Like much previous work [12, 15, 28], we get the test images from typical image datasets such as Cablecar, Pepper, and Baboon. These images are resized to produce new images with different sizes. Then, we obtain the compressed images by applying our proposed method and the sequential algorithm, respectively. The corresponding PSNR and SSIM of these compressed images are computed to evaluate the image quality.

Table 2 presents the experimental results on the test images. When an image is compressed by the same entropy coding method, the PSNR values differ significantly due to different image formats. The main reason is that there is more information loss caused by critical down-sampling operation on chrominance components. However, there is little difference between SSIM values, which indicates that the compression rate can be further improved without affecting the subjective perception of the image by the human eye when the 4:2:0/4:2:2 image format is

Table 2 Compressed image quality comparison ($Q=50$)

Evaluation index	Image formats	Coding method	Image Dimension(width × height)				
			768 × 512	1200 × 768	1440 × 900	1760 × 1100	2560 × 1600
PSNR	4:4:4	CPU Serial	30.784	32.389	32.489	33.509	35.108
		GPU Parallel	30.791	32.397	32.640	33.527	35.220
	4:2:2	CPU Serial	29.698	30.914	31.895	32.098	33.825
		GPU Parallel	29.884	31.281	31.901	32.477	34.096
	4:2:0	CPU Serial	29.263	30.378	30.882	31.820	33.443
		GPU Parallel	29.315	30.466	30.977	31.884	33.529
SSIM	4:4:4	CPU Serial	0.9269	0.9308	0.9325	0.9358	0.9490
		GPU Parallel	0.9272	0.9309	0.9330	0.9362	0.9499
	4:2:2	CPU Serial	0.9270	0.9298	0.9309	0.9349	0.9486
		GPU Parallel	0.9271	0.9305	0.9312	0.9354	0.9487
	4:2:0	CPU Serial	0.9254	0.9301	0.9299	0.9347	0.9482
		GPU Parallel	0.9256	0.9304	0.9307	0.9349	0.9485

employed. More importantly, the compressed image quality obtained by parallel and sequential methods is almost the same. The visualization results of some samples with different formats shown in Fig. 11 also prove the views expressed above. Therefore, we can conclude that the compressed image obtained by our proposed method has the same quality as the sequential algorithm.

4.3 Relationship between consuming time and image size

The image size directly affects the number of 8×8 blocks that require to be processed, and the consuming time of entropy coding will also change. In this subsection, we keep the other conditions unchanged and merely change the image size to explore the relationship between the image size and consuming time. Specifically, the source images with different sizes are compressed by four different methods, including the sequential algorithm, NPP v10.2 [29], the reproduced Shan's method [28], and our EPEnt. The main characteristics of the previous methods are described as follows:

Sequential algorithm All the steps of image compression meet the standard of JPEG and are implemented based on CPU.

Shan's method [28] It is a semi-parallel version for the JPEG image compression algorithm that has implemented the entropy coding by the coordinated action of GPU and CPU.

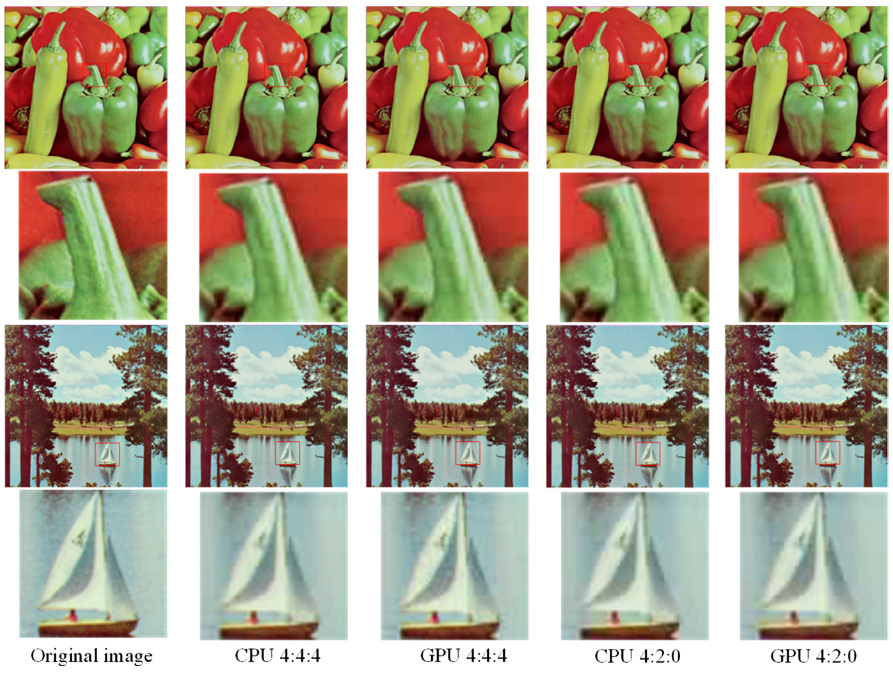


Fig. 11 Original and compressed images using different methods ($Q=50$)

NPP v10.2 [29] It is the first efficient and classic parallel version for the JPEG image compression algorithm, which has implemented the whole procedure based on GPU. Although it has achieved a faster processing speed than previous methods, the entropy coding stage is still a performance bottleneck.

During the compression, the total compression time of an image is measured. Precisely, the entire compression time of parallel implementations consists of the kernel execution, and data transfer between the host and device. The execution time of entropy coding is measured to verify the performance of our proposed method.

Through the source images, we obtain the compressed images with three different formats (i.e., 4:2:0/4:2:2/4:4:4). Figure 12 summarizes the compression time changed with different image sizes and formats. From it we can draw that as the image size becomes larger, the compression time of sequential implementation increases significantly. Owing to the incomplete parallelization, it is found that the execution time of Shan's method has the same change laws with sequential implementation. Moreover, the transmission of entropy coding intermediate data between the host and device increases the time consumption, which affects the overall performance. Although the consuming time of EPEnt and NPP v10.2 also increases, the change is very gentle. It is noteworthy that when the NPP v10.2 is employed to complete the image compression, the execution time of entropy coding still accounts for a large proportion of the whole process. The results demonstrate that the execution time will be significantly shortened when our proposed method is adopted. Our

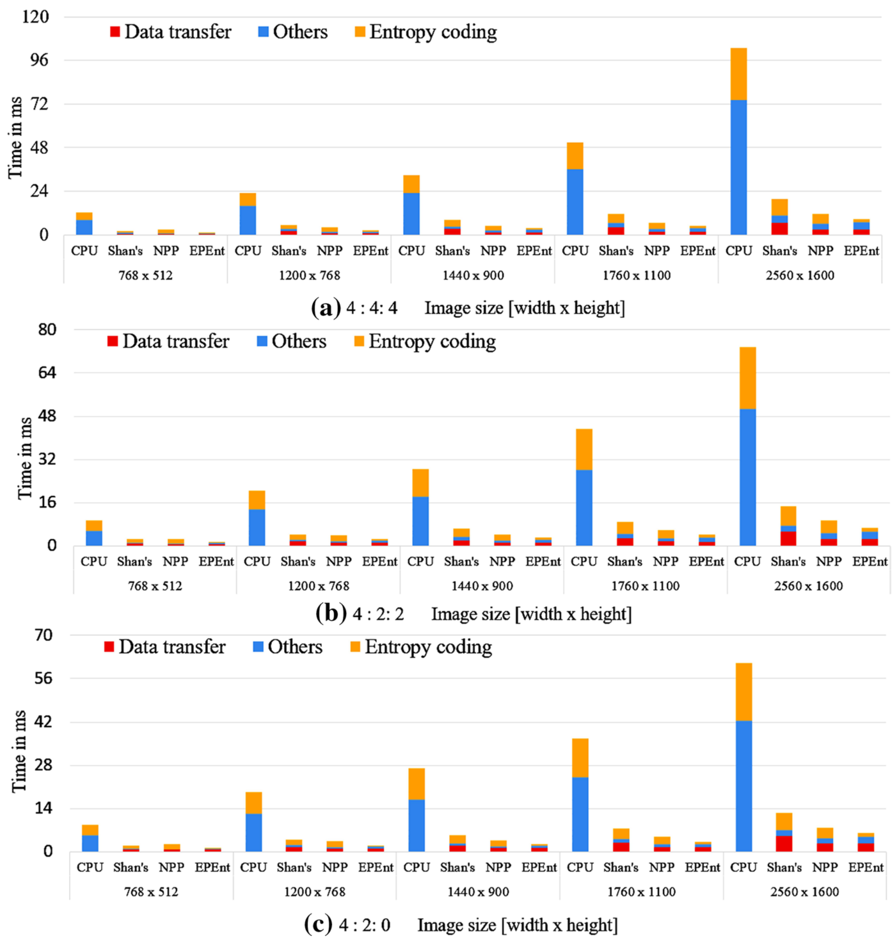


Fig. 12 The overall consuming time with different image size in **a** 4:4:4 formats, **b** 4:2:2, and **c** 4:2:0 formats, comparing parallel with sequential consuming time. ($Q=50$)

proposed EPEnt has the lowest time complexity and achieves the best performance than other advanced methods.

To further evaluate the performance of EPEnt, the CUDA speedups relative to the sequential algorithm based on CPU are calculated, which is described below:

$$\text{speedup} = \frac{t_s}{t_p} \tag{8}$$

where t_s is the sequential consuming time and t_p is the parallel consuming time for CUDA implementation. Firstly, we calculate the parallel compression speedups, where t_p is the total compression time, including kernel execution and data transfer. Then, parallel entropy coding speedups are calculated where t_p merely represents the consuming time of entropy coding. The performance improvements over the state-of-the-art parallel method (NPP) are also reported in this paper, which can be calculated as follows:

$$\text{Improvement} = \left(\frac{t_{pN}}{t_{pE}} - 1\right) \times 100\% = \frac{\text{speedup}_{pE} - \text{speedup}_{pN}}{\text{speedup}_{pN}} \times 100\% \tag{9}$$

where t_{pN} and t_{pE} denote the execution time of NPP and our method, respectively. $speedup_{pE}$ and $speedup_{pN}$ are the speed ratios of the two parallel methods relative to sequential implementation.

Figure 13 presents the compression speed gains for CUDA parallel programs over the sequential CPU-based program. As the image size increases, the GPU-based methods achieve higher speedups apart from Shan's method due to its incomplete parallelization. Particularly, when encoding images in 4:4:4 format, the speedup ratio is always higher than that in 4:2:2 format and 4:2:0 format. In EPEnt, the maximum speedups of entropy coding obtained with a large-size image of 2560×1600 (19.0 in 4:4:4 format, 16.0 in 4:4:4 format, and 16.8 in 4:2:0 format). Even compared with start-of-the-art NPP v10.2, the speedup of EPEnt increases by at least 208.4%

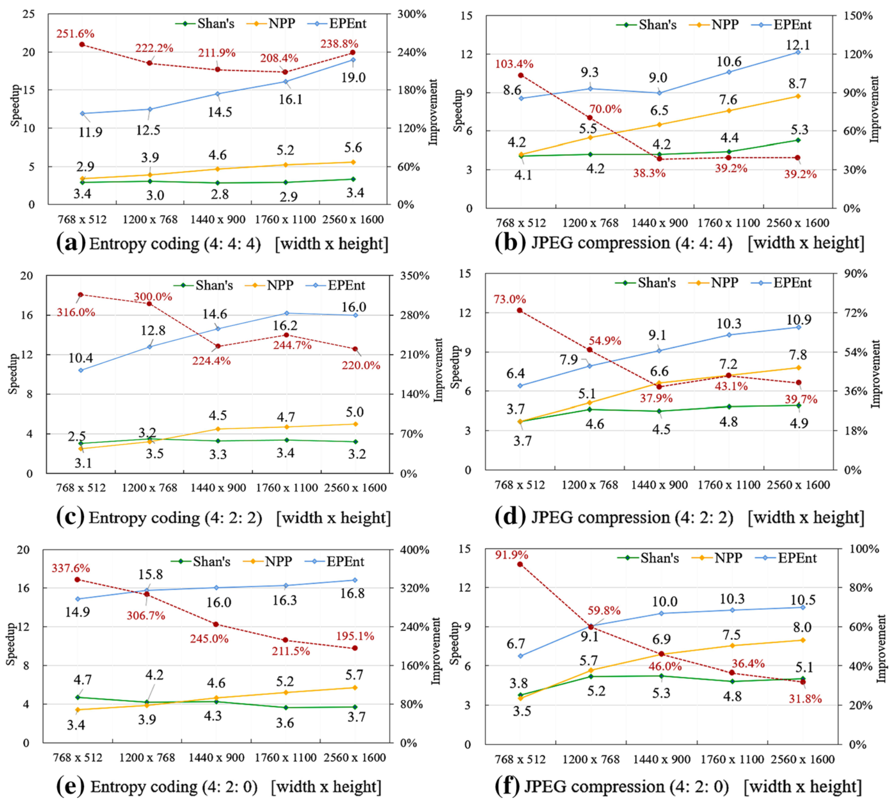


Fig. 13 Speedup of CUDA parallel entropy encoding relative to sequential algorithm at different image size ($Q = 50$). Compared with the state-of-the-art NPP v10.2, the improvement of our method is also reported

in in 4:4:4 format, 220.0% in 4:2:2 format, and 195.1% in 4:2:0 format, respectively. Therefore, owing to the limitation of entropy coding, the maximum speedups of the two other GPU-based methods can merely achieve up to 8.7 × and 5.3 × speedups when performing their JPEG compression algorithms entirely. More importantly, we can observe that the JPEG compression process efficiency has increased by at least 30% compared with the state-of-the-art NPP v10.2 in term of speedup ratio.

4.4 Relationship between consuming time and image quality

The standard quantization tables and image quality value Q (1–100) are introduced to produce quantization coefficients, which are the key factors to control the quality of compressed image. As the value Q increases, the quantization coefficients will be reduced, and the image quality will increase accordingly. To quantitatively analyze the effect of the value Q, we choose images of size 2560×1600 to obtain compressed images using our proposed method and the sequential algorithm, respectively. The images with different qualities can be generated when we change the value Q. Afterward, we calculate the PSNR and SSIM of these compressed images. Table 3 illustrates that the compressed images obtained by both methods can guarantee the same quality and are unaffected by the variation of Q.

The number of non-zero coefficients in each 8×8 block grows as the value Q increases, which leads to much more cumbersome coding tasks and prolonged time consumption during the entropy coding stage. Similarly to Sect. 4.2, we evaluate the results obtained from our proposed method and compare them with the other methods in term of compression time. The comparison also involves three other methods: the sequential algorithm, NPP v10.2, and Shan's method.

Figure 14 shows our experiments' execution time and provides a clear comparison among those parallel and sequential algorithms. The total execution time is the

Table 3 The effect of value Q on image quality

Evaluation index	Image formats	Coding method	Image quality value Q				
			50	60	70	80	90
PSNR	4:4:4	CPU Serial	35.108	35.461	36.338	38.458	40.987
		GPU Parallel	35.220	35.542	36.456	38.671	41.060
	4:2:2	CPU Serial	33.825	34.836	35.747	36.993	38.845
		GPU Parallel	34.096	35.149	35.947	37.012	38.925
	4:2:0	CPU Serial	33.443	34.102	34.782	35.976	37.423
		GPU Parallel	33.529	34.179	34.941	36.123	37.559
SSIM	4:4:4	CPU Serial	0.9490	0.9536	0.9591	0.9683	0.9815
		GPU Parallel	0.9499	0.9543	0.9597	0.9691	0.9823
	4:2:2	CPU Serial	0.9486	0.9529	0.9588	0.9671	0.9811
		GPU Parallel	0.9487	0.9532	0.959	0.9673	0.9816
	4:2:0	CPU Serial	0.9482	0.9525	0.9577	0.9667	0.9809
		GPU Parallel	0.9485	0.9531	0.9586	0.9672	0.9812

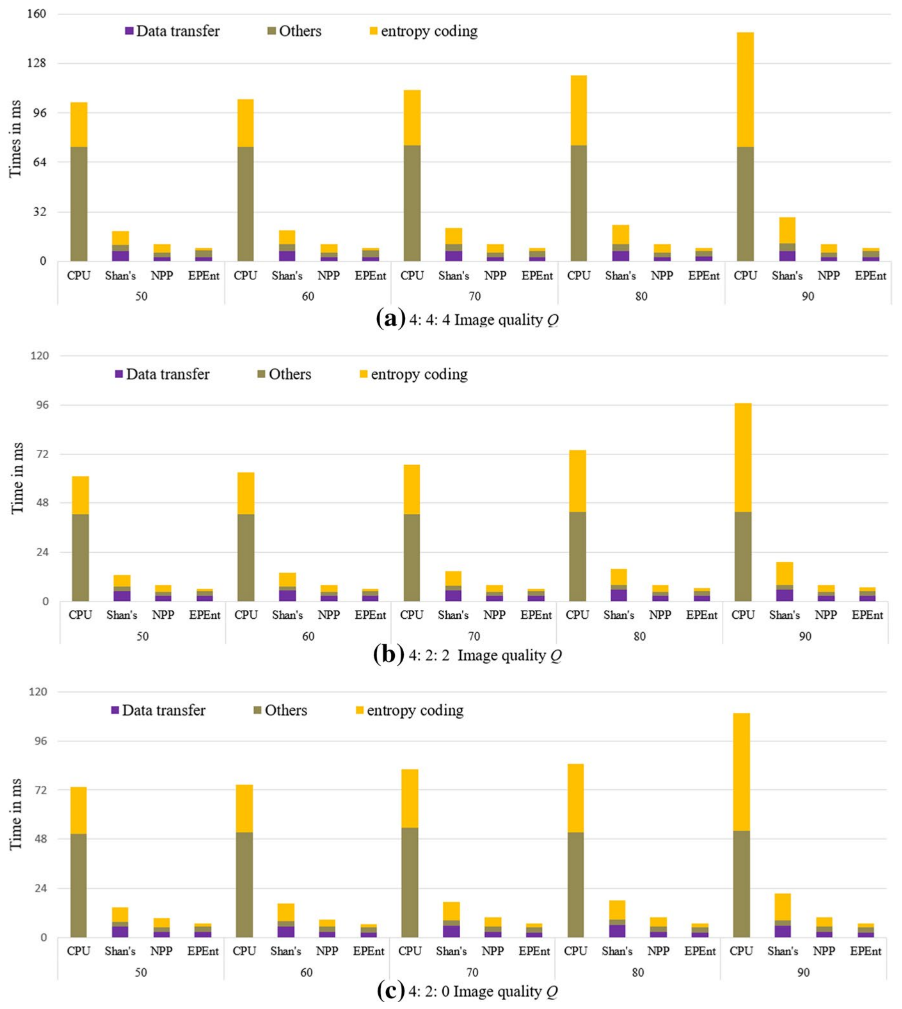


Fig. 14 The overall consuming time with image quality value Q in **a** 4:4:4 formats, **b** 4:2:2, and **c** 4:2:0 formats, comparing parallel with sequential consuming time

accumulated execution times of kernels and data transmission between the CPU and GPU. As shown in the graph, when the value Q is raised in different formats (i.e., 4:4:4/4:2:2/4:2:0), the GPU-based implementations achieve significantly better performance than the sequential CPU-based method of total execution time. The performance of these GPU-based methods is very promising. Especially when our proposed EPEnt is applied to the JPEG compression algorithm, considerable reductions in execution time will be achieved. It further proves that our proposed method has effectively reduced the time complexity.

Similarly, we compute the speedups of three parallel methods by utilizing the formulas defined in (8). The performance boost of our method relative to NPP v10.2

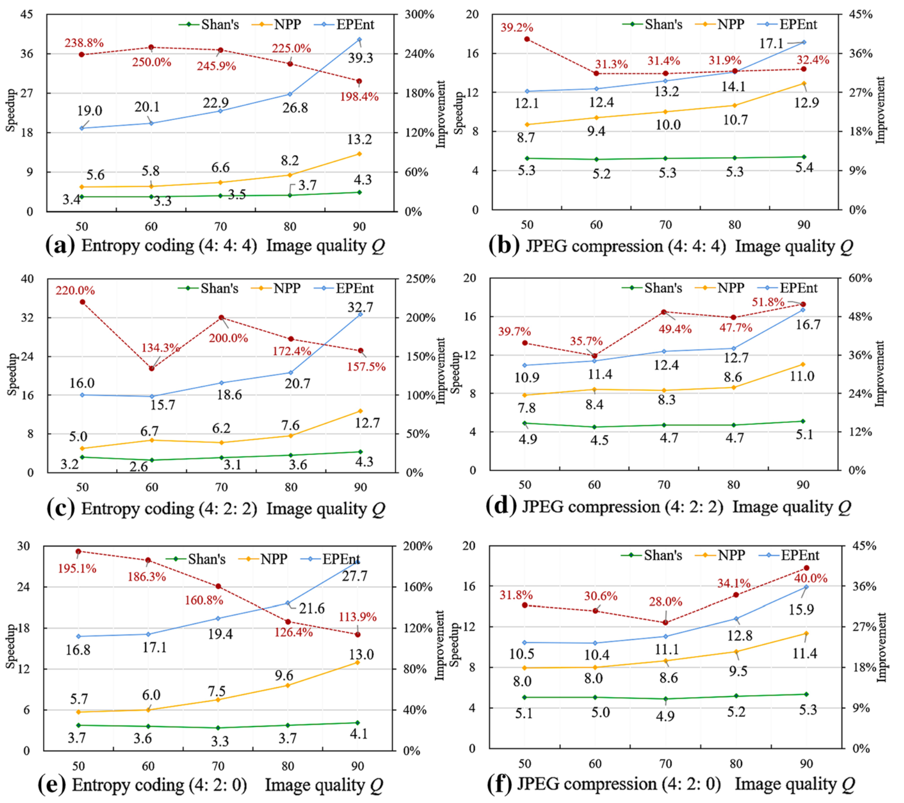


Fig. 15 Speedup of CUDA parallel entropy encoding relative to sequential algorithm at different Q. Compared with the state-of-the-art NPP v10.2, the improvement of our method is also reported

is also quantified through formula (9). In Fig. 15, we report the result that summarizes the performance of three parallel methods with regard to the entropy coding and the entire process. From the figure, we can conclude that the speedups achieved by the three parallel methods are growing with the value Q. To be specific, when images are encoded in 4:4:4 format, the speedups are invariably higher than that of the 4:2:0 format and 4:2:2 format. Moreover, the maximum speedups of the whole compression can be achieved in all formats (e.g., 4:4:4/4:2:2/4:2:0) when our proposed EPEnt is adopted. Even compared with the state-of-the-art NPP v10.2, the performance improvement of our method can achieve at least 28% during the JPEG compression process. Therefore, our proposed parallel entropy coding method plays an influential role in parallel JPEG image compression.

4.5 Impact of coding method on efficiency

In Sect. 3.2.1, the coding kernels functions are, respectively, designed for both the Y component and CbCr component. Here, we conduct some experiments to verify our

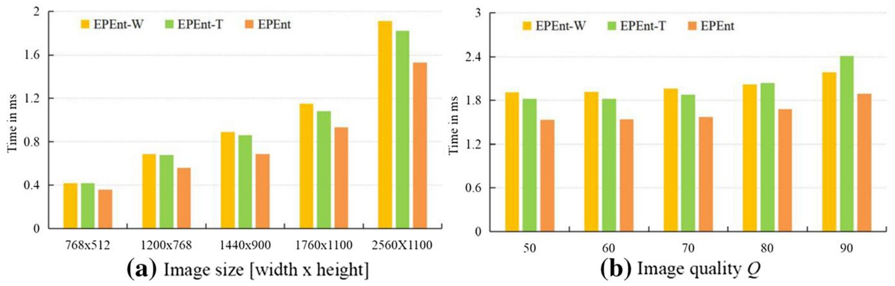


Fig. 16 The comparison of different strategies in the coding phase

method. To be specific, three strategies are designed to encode the same image data with 4:4:4 format: the thread-based method (EPEnt-T), the warp-based method (EPEnt-W), and the combination of thread-based and warp-based methods (EPEnt). Figure 16 presents the coding results with regard to execution time. From it, we observe that our proposed EPEnt obtains the best performance. The main reason is that our EPEnt makes full use of the data feature of both luminance component and chrominance components and maximizes the resource utilization of GPU.

5 Conclusion

In this paper, an efficient parallel entropy coding method for JPEG image compression is presented based on CUDA architecture. The proposed method has three phases: the coding phase, the shifting phase, and the stuffing phase. In the coding phase, 8×8 blocks are coded in parallel. For the different characteristics of the Y component and CbCr component, two different kernel functions are designed to encode them. In the shifting phase, the code results of all 8×8 are shifted to ensure that the encoding results of adjacent blocks can be stitched together continuously. The final output stream can be produced in the stuffing phase.

We have tested the proposed method with a large set of typical images with different sizes and formats. Experimental results show that our proposed method has achieved significantly better performance than the other methods. Consequently, our proposed method is readily suitable for real-time and/or near real-time processing applications. We will continue to optimize the parallel JPEG image compression algorithm for higher speed and better image quality in our future work. Afterward, we will extend our work and implement other parallel compression algorithms on CUDA, such as JPEG 2000 and MPEG.

Acknowledgements The authors would sincerely like to thank the editor and anonymous reviewers for their detailed review.

References

1. Aguilar AH, Bonilla-Robles JC, Díaz JCZ et al (2019) Real-time video image processing through GPUs and CUDA and its future implementation in real problems in a Smart City. Int J Combinat Optim Prob Inform 10(3):33–49
2. Haiyan Zhang A (2002) Image compression. Technology 14(7):831–835
3. Li J, Wu J, Jeon G et al (2020) GPU acceleration of clustered DPCM for lossless compression of hyperspectral Images. IEEE Trans Industr Inf 16(5):2906–2916
4. Wallace GK (1991) The JPEG still picture compression standard. Commun ACM 34(4):30–44
5. Tadisetty S (2019) A novel ortho normalized multi-stage discrete fast Stockwell transform based memory-aware high-speed VLSI implementation for image compression. Multim Tools Appl 78(13):17673–17699
6. Salah A, Li K, Hosny KM et al (2020) Accelerated CPU–GPUs implementations for quaternion polar harmonic transform of color images. Futur Gener Comput Syst 107:368–382
7. Spiliotis IM, Bekakos MP, Boutalis YS (2020) Parallel implementation of the image block representation using OpenMP. J Parall Distrib Comput 137:134–147
8. Hosny KM, Salah A, Saleh HI et al (2019) Fast computation of 2D and 3D Legendre moments using multi-core CPUs and GPU parallel architectures. J Real-Time Image Proc 16(6):2027–2041
9. Yuan Y, Yang X, Wu W et al (2019) A fast single-image super-resolution method implemented with CUDA. J Real-Time Image Proc 16(1):81–97
10. Alqudami N, Kim SD (2016) OpenCL-based optimization methods for utilizing forward DCT and quantization of image compression on a heterogeneous platform. J Real-Time Image Proc 12(2):219–235
11. Ghetia S, Gajjar N, Gajjar R (2013) Implementation of 2-D discrete cosine transform algorithm on GPU. Int J Adv Res Electric Electron Instrum Eng 2(7):3024–3030
12. Haweel RT, El-Kilani WS, Ramadan HH (2016) Fast approximate DCT with GPU implementation for image compression. J Vis Commun Image Represent 40:357–365
13. Obukhov A, Kharlamov A (2008) Discrete cosine transform for 8x8 blocks with CUDA. NVIDIA white paper
14. Tokdemir S, Belkasim S. Parallel processing of DCT on GPU. 2011 Data Compression Conference. IEEE, 2011: 479–479
15. Shan R, Zhou X, Wang CY et al (2016) All phase discrete sine biorthogonal transform and its application in JPEG-like image coding using GPU. TIIS 10(9):4467–4486
16. Wang C, Shan R, Zhou X (2015) APBT-JPEG image coding based on GPU. KSII Trans Int Inform Syst (TIIS) 9(4):1457–1470
17. Shatnawi MKA, Shatnawi HA A performance model of fast 2D-DCT parallel JPEG encoding using CUDA GPU and SMP-architecture. 2014 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2014: 1-6
18. Liu D, Fan XY. Parallel program design for JPEG compression encoding. 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery. IEEE, 2012: 2502–2506.
19. Enfedaque P, Auli-Llinas F, Moure JC. Strategies of SIMD computing for image coding in GPU. 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). IEEE, 2015: 345–354.
20. Balevic A. Parallel variable-length encoding on GPGPUs. International Conference on Parallel Processing, 2009: 26–35.
21. Patel P, Wong J, Tatikonda M, et al. JPEG compression algorithm using CUDA. Department of Computer Engineering, University of Toronto, Course Project for ECE, 2009, 1724.
22. Zhang M, Zhang J, Qiu X (2017) Parallel design and implementation of JPEG compression algorithm based on OpenCL. Comput Eng Sci 39(5):855–860
23. Rahmani H, Topal C, Akinlar C (2014) A parallel Huffman coder on the CUDA architecture[C]. In: IEEE Visual Communications and Image Processing Conference, vol 2014. IEEE, pp 311–314
24. Sudarshan ESC and Chigarapalle S, 2017 A compact parallel Huffman entropy coding technique on GPGPU using CUDA. ARPN J Eng Appl Sci 7111–7118.
25. Single pass prefix sum in a vertex shader. U.S. Patent Application 16/007,893. 2019.
26. M. Harris, S. Sengupta, J. D. Owens, H. Nguyen. Parallal prefix Sum (Scan) with CUDA, in: GPU Gems 3 Part VI: GPU Computing, Addison Wesley, 2007: 851–876.

27. Sengupta S, A. E Lefohn, J.D. Owens. A work-efficient step-efficient prefix sum algorithm, in: Workshop on Edge Computing Using New Commodity Architectures, 2006.
28. Shan R, Wang C, Huang W, Zhou X (2015) DCT-JPEG image coding based on GPU. Int J Hybrid Inform Technol 8(5):293–302
29. NVIDIA CUDA C++ Programming Guide, 10.2, 2018
30. Harris M. Optimizing parallel reduction in cuda, [online] Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
31. Sodsong W, Jung M, Park J, et al. JParEnt: Parallel entropy decoding for JPEG decompression on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 2017, 29(15).
32. Hore A, Ziou D. Image Quality Metrics: PSNR vs. SSIM. International Conference on Pattern Recognition, 2010: 2366–2369.
33. Pereira AD, Ramos L, Goes LF et al (2015) PSkel: A stencil programming framework for CPU-GPU systems. Concurren Comput Prac Exp 27(17):4938–4953
34. Tian J , Rivera C , Di S , et al. Revisiting huffman coding: toward extreme performance on modern GPU Architectures[C]// The 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2020.
35. Yamamoto N, Nakano K, Ito Y, et al. Huffman Coding with Gap Arrays for GPU Acceleration[C]//49th International Conference on Parallel Processing-ICPP. 2020: 1–11.