Image Codec Parallelization: A Case Study of libjpeg

Sun, Wen-Lin
wlsun.eed06g@nctu.edu.tw
Institute of Electrical and Computer Engineering, NCTU
Student ID: 0680707

Wang, Sheng-Kai
kevin9101393.eed07g@nctu.edu.tw
Institute of Electrical and Computer Engineering, NCTU
Student ID: 0750724

Hsieh, Bing-Ruei
aphonethree.eed03@nctu.edu.tw
Institute of Electrical and Computer Engineering, NCTU
Student ID: 0760234

ABSTRACT

JPEG is a lossy-compression image format that has been widely used in decades and its acceleration of compression process may benefits numerous users and other depending technologies (e.g. digital device with photographic functions, social media platform, etc.). To accelerate the compression process, we design our solutions with data-level parallelism (`CUDA` and function-level parallelism (`OpenMP`) on the hot spots found by the `grpof` profiler. We implement our solutions and evaluate them with different sizes of images. Also, we discuss and compare our solutions to the SIMD approach developed by `libjpeg-turbo`. Moreover, with the further discussions of our experiments, we comes to a point, called workload-input dependency, which assists researchers in choosing between parallelism technologies for image codec parallelization.

KEYWORDS

image codec parallelization, JPEG, SIMD, OpenMP, CUDA

1 INTRODUCTION

Various image file formats have been introduced to the public, including JPEG, TIFF, GIF, PNG, etc. Each of them is designed for some special use case like photo shooting, scanning, printing and internet uses. Among them, Joint Photographic Experts Group (JPEG), whose file extension is ".jpg" and is one of the most popular and well-known image file format. JPEG is a lossy compression algorithm and is widely used on Internet and image databases [16]. Although JPEG has a lossless compression version [14], the lossy one is the most popular and our focus here. Digital cameras and websites, such as Google Photos [2], often utilize JPEG since it heroically compresses the image to a much smaller size.

`libjpeg` [3] is a widely used free library for JPEG image compression/decompression and it implements the compression and decompression processes with a codec based on the JPEG standard in C programming language [4]. Both the compression and decompression processes are divided into 6 phases. After profiling `libjpeg` with `gprof`, a widely used profiler [13] on UNIX system, we discover that many hot spots are potential to deploy parallelization technique to speed up the compression process. According to the profiling result, we choose three hot spots in two phases, color-space conversion (one hot spot) and forward Discrete Cosine Transform (DCT) (two hot spots). We deploy two different parallelism techniques, function-level parallelism (`OpenMP`) and data-level parallelism (`CUDA`), on the selected hot spots, and make the standard JPEG codec implementation more efficient. We evaluate our solutions and compare the evaluation results to `libjpeg-turbo` [5], an improved version of `libjpeg` that speeds up the baseline JPEG algorithm with SIMD instructions [6]. For the color-space conversion phase and forward DCT phase, we provide three key points and three stages to evaluate and discuss the solutions, respectively. Firthermore, with the further discussions of our experiments, we summarize a point of view, which assists researchers in choosing between parallelism technologies for image codec parallelization, called workload-input dependency.

The proposal is organized follows. In Section 2, we state our problem and detail our proposed solutions. Then, we introduce `libjpeg-turbo` and its solution in Section 3. Next, we describe our experimental methodology and make thorough discussions on experimental results in Section 4 and Section 5, respectively. After that, we conclude our project in Section 6.

2 PROPOSED SOLUTIONS

In this section, we present our solutions of parallelizing `libjpeg`, which is wrote in C language. First, in Section 2.1, we give an overview of `libjpeg` and analyze it with the profiling tool `gprof`. Then, based on previous analysis, we parallelize the selected hot spots with function-level and data-level parallelisms independently. We divide our solutions into two parts with the phases of the compression process they belong to: one is color-space conversion Parallelization (Section 2.2), and the other one is forward DCT Parallelization (Section 2.3).

2.1 Problem Overview

The main flow of BMP-JPEG conversion realized in `libjpeg` are illustrated in Fig. 1. In accordance to the compression process, a BMP file stored in *RGB* color space is transformed to YC_BC_R color space in the color-space conversion phase. Then, in the down-sampling phase, the color-space-transformed image is down-sampled and split into blocks of $8x8$ pixels, called Minimum Coded Units (MCUs). Next, each MCU is deployed forward DCT, which transform the MCU from time-domain to frequency-domain representation and determined the DCT coefficients for further quantization. In the qantization phase, the DCT coefficients are then quantized with quantization matrix that controls the compression ratio. In JPEG standard, entropy coding scheme is used after quantization for lossless data compression. There are two common entroy coding techniques, Huffman coding and arithmetic coding, and the former is implemented in `libjpeg`. The Huffman coding progress generates the Huffman table for decoding in decompression process. Otherwise, the decompression process is a reverse process of compression with looking up but building the tables.

All the phases in the process of compression and decompression include large amount of matrix calculations and table look-ups, and, in most of the phases, these actions are done with blocks (MCUs) serially. According to our survey on theoretical algorithms used in

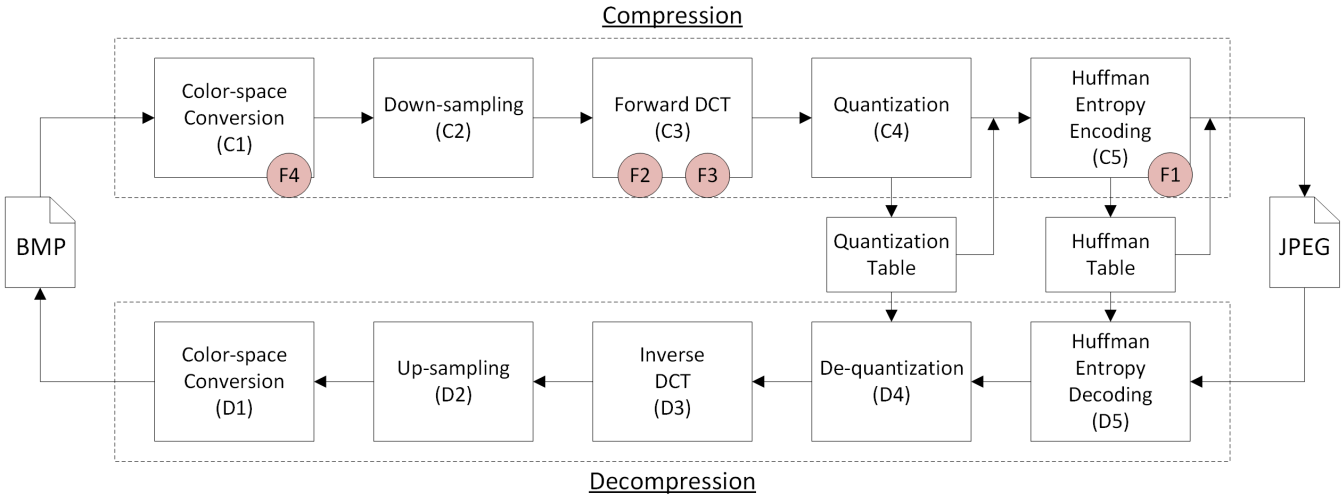

Figure 1: The BMP-JPEG conversion flow in `libjpeg`.

Table 1: Top-6 hot spots of `cjpeg`.

Symbol	Time (%)	Function Name	Phase
*	42.33	preload_image	read file
F1	14.16	encode_mcu_huff	C5
F2	12.07	jpeg_fdct_16x16	C3
F3	9.99	jpeg_fdct_islow	C3
F4	9.69	rgb_ycc_convert	C1

JPEG and practical source code of `libjpeg`, most of the calculations can be done independently. Therefore, in this project, we dedicate to parallelization of the time-consuming functions (hot spots) in the compression process.

To find out the hot spots of the in the compression process, we use `gprof` to profiling `cjpeg`. The profiling result of `cjpeg` is shown in Table 1. The testing image file is about 909 MB, which is big enough to make the total processing time long enough for `gprof` to profiling clearly. In Table 1, we list the top 5 hot spots, including the function name, the percentage of the total time that was spent in the function and its children, and the corresponding phase to the compression process (red spots in Fig. 1). According to the profiling results, except for the I/O operation (read file) and the fifth phase (C5), Huffman Entropy Encoding, in which each iteration is not independent, `cjpeg` costs most of its running time on the first (C1) and the third (C3) phases. Therefore, we choose the remaining hot spots (F2-4), which consume about 32% of execution time, to be parallelize. We analyze the data dependency of each hot spot and parallelize it with both function-level paralellism (`OpenMP` [7] [8]) and data-level paralellism (`CUDA` [9]).

2.2 Color-Space Conversion Paralellization

2.2.1 Background. Color-space conversion is a translation process that translates color representation from a basis to another [10]. In JPEG specification, YC_BC_R is used, in which Y represents the luma component that, and C_B and C_R represent blue-difference and red-difference chroma components, respectively. In comparison to Y, Y' is commonly used in implementation. Y' represents luminance, which means that light intensity is non-linearly encoded based on gamma-corrected *RGB* primaries [11]. Eq.(1) shows the transfer equation from gamma-corrected *RGB* to YC_BC_R,

$$\begin{aligned} Y' &= 0.29900 \cdot R' + 0.58700 \cdot G' + 0.11400 \cdot B') \\ C_B &= 128 + (-0.16874 \cdot R' - 0.33126 \cdot G' + 0.50000 \cdot B') \\ C_R &= 128 + (0.50000 \cdot R' - 0.41869 \cdot G' - 0.08131 \cdot B') \end{aligned} \quad (1)$$

where R', G' and B' represents gamma-corrected R, G and B in *RGB* color space, respectively.

2.2.2 `libjpeg`'s Solution: Look-up Table. `libjpeg` accelerates the color-space conversion process with a pre-built table, called look-up table. The look-up table is built once before calling F4. The table is composed of all the possible results of the multiplications in Eq.(1), that is, 8 different scalars multiply 256 values (8 bits) of gamma-corrected *RGB*. With the look-up table, the conversion process is simplified to summation of looked-up values from the table, so the repeat multiplication time is saved.

2.2.3 Our Solution. Based on the look-up table approach implemented in `libjpeg`, We deploy function-level parallelism (`OpenMP`) and data-level parallelism (`CUDA`) with work-sharing philosophy on the function `rgb_ycc_convert` (F4). F4 does the row-wise color-space conversion from input image's *RGB* color space to output image's YC_BC_R color space. As the result, the workload of F4 is depended on the length of rows of the input image. We detail both the methodologies of `OpenMP` and `CUDA` separately:

- OpenMP: According to our analysis of F4, the work of F4 includes looking up the look-up table and summation the look-up results on the basis of Eq. (1). The work has no data-dependency to previous iterations, so it is able to be done independently; moreover, there is no critical section in F4 and the calculation workload are exactly the same in each

call (same width of rows). With above statement, we equally share the work with dividing the rows to the number of threads, and each thread is lock-free and has no dependency with others.

- CUDA: To make most of the calculations in the conversion been done on a GPGPU, we implements the color-space conversion with `CUDA C/C++`. Reducing the data transmission between host (CPU) and device (GPU) is critical in heterogeneous computing. As the result, to avoid the repeatedly data access to host memory while looking up the look-up table, we allocate device memory and copy the whole look-up table to the device memory once before we start to calculate the input row. Also, the device memory for input row is allocated and filled with the data of input row. Then, the input row is divided equally to blocks of threads. The number of block and the number of threads per block depend on the selected GPU specification. After the input row is conversed, the output row will transmit from the device to the host and the allocated memory spaces on the device will be free.

2.3 Forward DCT Parallelization

2.3.1 Background. In `libjpeg` compression algorithm, the downsampled YC_BC_R color space image goes through a two-dimensional (2-D) forward DCT process, which transform the image from time-domain representation to frequency-domain representation. Instead of applying the transformation directly on the entire image, the compression algorithm first splits the input image into 8×8 or 16×16 MCUs. Then, perform 2-D forward DCT on each MCUs respectively. The general direct 2-D forward DCT equation is shown in Eq.(2),

$$F(u,v) = (\frac{2}{N})^{\frac{1}{2}}(\frac{2}{M})^{\frac{1}{2}} \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} \Lambda(i).\Lambda(j).\cos[\frac{\pi.u}{2.N}(2i+1)]\cos[\frac{\pi.v}{2.M}(2j+1)].f(i,j) \quad (2)$$

where

$$\Lambda(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & for \xi = 0 \\ 1 & otherwise \end{cases}$$

and u and v are the horizontal and vertical spatial frequencies, respectively.

2.3.2 `libjpeg`*'s Solution: Indirect Method.* `libjpeg` apply an indirect method proposed in [15]. In the indirect method, no matter what the size the input MCU is, 1-D DCT is performed row-by-row, and then performed column-by-column. The indirect method is mathematically equivalent to 2-D DCT in Eq.(2) but less complex and faster for implementation.

2.3.3 Our Solution. This indirect method gives us a chance to parallelize it because the 1-D DCT process on each row and each column are all independent. Note that all processes of row-wise 1-D DCT need to be done before the processes of column-wise 1-D DCT.

The other chance we can deploy parallelization to is how the entire image goes through the 2-D DCT process. The key point is that, after the image is split into MCUs, each MCU is processed independently. As the result, we can make the process parallelly executed with no racing condition. We deploy function-level parallelism (`OpenMP`) on the 2-D DCT of each MCU, and deploy data-level parallelism (`CUDA`) on the 2-D DCT of the entire image. Since each MCU contains too little data to be parallel executed, and it does not conform to the design purpose of GPU and CUDA, we do not adopt `CUDA` on the 2-D DCT of each MCU.

Based on the description above, we divide our solution into three stages. In the first stage, we use `OpenMP` to perform function-level parallelism on the forward DCT of each MCU (in F2 and F3), which we call the inner loop of the process. Since each row-wise and column-wise 1-D DCT is independent, we distribute the workload equally to each thread and this can be easily done using a one-line statement in OpenMP. An extra note is that all row-wise 1-D DCTs need to be done before any column-wise 1-D DCTs, so there are two parallel regions in the process.

In the second stage, we use `OpenMP` to parallelize the forward DCT between all MCUs (the functions who call F2 and F3) and call this stage the outer loop of the process. Since foward DCT on each MCU is independent, the parallelization process is pretty much the same as the one we did in the first stage. Each foward DCT is equally distributed to the threads to balance the workload.

In the third stage, we use `CUDA` to perform data-level parallelism on the outer loop of the process. To reduce the memory allocation time and also the data transferring time between the main memory and the device memory, we allocate the memory for the entire image and transfer it in and out the device all at once. Since number of threads per block does not greatly affect our results, we choose the maximum number (1024) of threads it allows and the results are all based on this setting.

The experimental results are discussed in Section5.2.

3 RELATED WORK

`libjpeg-turbo` is a JPEG image codec that boost the compression/decompression process performance by using Single Instruction, Multiple Data (SIMD) instruction set extensions. SIMD extensions allow a single instruction simultaneously processed by multiple processing elements in the same cycles and are known as data-level parallelism. `libjpeg-turbo` is based on `libjpeg` and declares to speed up the whole JPEG compression process 2-6x in comparison with `libjpeg` [1]. In contrast, the APIs we utilized exploit both data-level (`CUDA`) and function-level parallelism (`OpenMP`). We compare and discuss our approaches to `libjepg-turbo`'s approach in Section 5. In this section, we note the key points of `libjpeg-turbo`'s solution in the selected hot spots, F3 and F4 (F2 is merged into F3). To realize `libjpeg-turbo`'s solution, we trace the source code written in Streaming SIMD Extensions 2 (SSE2) for x86-64 platform.

3.1 `libjpeg-turbo`'s Color-Space Conversion Solution

Instead of `libjpeg`'s look-up table solution, `libjpeg-turbo` uses the 128-bit width registers to speed up the calculations, including both the multiplication and the addition, used in color-space conversion. According to Figure 2, in each iteration, `libjpeg-turbo` fills three 128-bit registers with 16 pixels from the input row and each pixel is 24-bit width with 8-bit values of R, G and B. To benefits

from the specialized instructions provided by SSE2, `libjpeg-turbo` rearranges the pixels of an input row from interleaved to odd-even-separated. Also, each R, G and B value of a pixel is widened from 8-bit width to 16-bit width, so the used number of 128-bit registers increases from three to six.

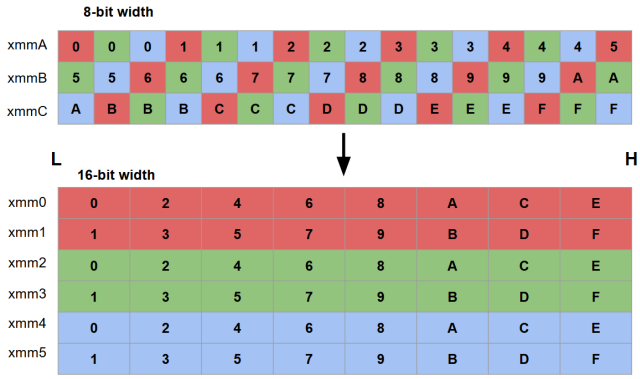

Figure 2: Pixel rearrangement in `libjpeg-turbo`.

Besides, `libjpeg-turbo` modifies the original color-space conversion equation (Eq.(1)) to Eq.(3) with division of the multiplication of G' term in Y''s calculation (underlined parts in Eq.(3)). This modification makes the total multiplication times become even and be able to make better use of some specialized instructions such as Multiply and Add Packed Integers (`pmaddwd`). These specialized instructions handle multiple data simultaneously, for example, the instruction `pmaddwd` does 8 multiplications and 4 additions in 1 MMX cycle, which equivalent C code takes 28 cycles [12].

$$\begin{aligned} Y' &= 0.29900 \cdot R' + \underline{0.33700 \cdot G'} + 0.11400 \cdot B' + \underline{0.25000 \cdot G'} \\ C_B &= 128 + (-0.16874 \cdot R' - 0.33126 \cdot G' + 0.50000 \cdot B') \\ C_R &= 128 + (0.50000 \cdot R' - 0.41869 \cdot G' - 0.08131 \cdot B') \end{aligned} \quad (3)$$

Although the single instruction speedup is high with SSE2, the overhead of rearrangement before actual calculation is innegligible. Also, `libjpeg-trubo` does all the calculations of Eq.(3), rather than using a pre-built look-up table like `libjpeg`. To make further discussion, we evaluate `libjpeg-turbo`'s solution of color-space conversion and detail it in Section 5.1 with the comparison to `libjpeg` and our solution.

3.2 `libjpeg-turbo`'s forward DCT solution

The implementation of forward DCT in `libjpeg-turbo` eliminates the use of F2 (jpeg_fdct_16x16). Hence, when the input image is split into MCUs, the size of MCUs become 8×8. In other words, only F3 (jpeg_fdct_islow) is implemented using SIMD extensions in `libjpeg-turbo`. According to the source code written in SSE2, F2 utilizes a set of 128-bit-wide registers called XMM. Elements in the 8×8 MCU are 16-bit-wide, and adopt the indirect method proposed by [15]. `libjpeg-turbo` packs 8 row- or column- elements in one XMM register and perform packed addition and packed multiplication on the register. With above approach, the 8 row-wise or column-wise 1-D DCT can be performed parallel via using these registers and instructions provided by SSE2 instruction set. `libjpeg-turbo`'s solution has several advantages over using OpenMP and CUDA. First of all, SIMD program only utilizes one core of CPU, which entirely eliminates the potential of false-sharing problem in multi-core parallelism. Second of all, SIMD is well-suited to small data parallelism, while other data-parallelism such as CUDA requires large amount of data to overcome the parallelization overhead. Using SIMD does have its own overhead, though, since it needs the user to explicitly move the data in and out the XMM registers, and as a result, it is more complex and error-prone for the users to deal with.

4 EXPERIMENTAL METHODOLOGY

To do fair experiments, we isolates the target functions and evaluate them with independent testing programs. To make the experiments more realistic, we collects necessary input data and output result to several testing files while executing `libjpeg`'s solution on testing images. As the result, the testing program can check the correctness and evaluate the performance of the target function by its testing files. The information of testing images are listed in Table 2 and the platform configurations of our experiment environment is shown in Table 3. The size of the large image is a hundred times larger than the small image because the width and the height of the large image are both ten times longer than the small image.

Table 2: Information of testing images.

	Size (MB)	Dimension	Depth (bit)
Large	909	23281×13656	24
Small	9.09	2328×1366	24

Table 3: Platform configuration of experiment environment.

	Configuration
CPU	Intel Xeon Gold 6136 ×48 (12 cores, 24 threads per CPU)
GPU	Tesla V100-PCIE-16GB
RAM	125GB
OS Kernel	Linux 4.4.0-116-generic

5 EXPERIMENTAL RESULTS

In this section, we benchmark and discuss our solutions. The experimental results are divided into two parts, color-space conversion and forward DCT. Also, we compare our solution with `libjpeg-turbo`, which is also a library derived from `libjpeg` (detailed in Section 3). Then, we summarize the experimental results and discussions.

5.1 Color-Space Conversion

To compare our solution to `libjpeg`'s and `libjpeg-turbo`'s solution, we evaluate these three solutions with a small image and a large image and the results are shown in Table 4 and Table 5, respectively. According to the experiment results listed in Table 4

and Table 5, we highlight and discuss the three key points while comparing our solution with others' solutions.

Table 4: Parallelization results of color-space conversion on small image.

	Time(us)	SD	Speedup	Efficiency
libjpeg (Serial)	6.71	0.83		
OpenMP (2)	5.51	0.63	1.2	0.6
OpenMP (4)	4.71	4.38	1.4	0.4
CUDA	3.9	0.97	1.7	
libjpeg-turbo	2.71	0.88	2.5	

Table 5: Parallelization results of color-space conversion on large image.

	Time(us)	SD	Speedup	Efficiency
libjpeg (Serial)	69.14	1.43		
OpenMP (2)	43.37	1.72	1.6	0.8
OpenMP (4)	22.37	0.91	3.1	0.8
CUDA	4.2	0.83	16.5	
libjpeg-turbo	23.46	0.93	2.9	

5.1.1 Key Point 1: Execution Time & Speedup. Except for the CUDA, the execution time of all the other solutions with large image is almost ten times larger than with small image. Also, the length of the input row of the large image is ten times of small image (mentioned in 4). Hence, this trend implies our declaration declared in Section 2.2.3 that the workload of color-space conversion depends on the length of the input row.

Otherwise, in comparison with others' solutions, our CUDA solution has better speedup on large image. The workload increment does effect the speedup of CUDA solution, but it depends on whether the increased workload is over the thread blocks provided by the target GPU. In common usage, a single is hard to fully fill the device memory on the GPU, so we comment that our CUDA solution is suitable for the usage with high-performance requirements on large image.

libjpeg-turbo's solution speeds up stably for both small and large image because its parallelism methodology is to reduce static ratio of calculation times by merge static number of calculation. In comparison to another data-level parallelism solution, CUDA, although CUDA solution has substantial speedup on large image, its speedup on small image suffers from the overhead of the data transmissions between the CPU host and the GPU device. The overhead can be ignored on large data, but on small data.

5.1.2 Key Point 2: Standard Deviation (SD). Most of the SDs are lower than 2.00 in our parallelization results, except for our 4-thread OpenMP solution with small image. The small image owns shorter length of the input row, which implies the shorter length of input row is assigned to a thread. The shorter input row assigned to a thread, the higher possibility for different threads to access the

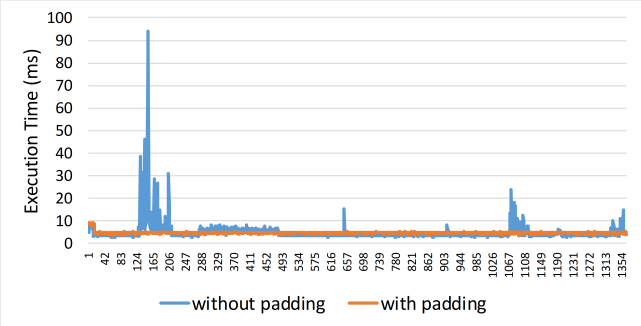

Figure 3: The execution time of each row - with and without padding comparison of 4-thread OpenMP solution on small image.

memory on the same cache line at the same time, that is, the false sharing problem. The false sharing problem is not clear in our 2-thread OpenMP solution with small image because there is only two threads sharing a input row, so the memory they accessed is far from each other with an enough distance. However, in our 4-thread OpenMP solution with small image, the false sharing problem is clear and is able to be found by checking the execution time of each row. Figure 3 illustrates the execution time of each row with and without padding on our 4-thread OpenMP solution with small image and the statistical results are shown in Table 6. According to Figure 3, there are several peaks before adding the padding to our 4-thread OpenMP solution, these peaks cause the high SD and implies the false sharing problem is happened. After adding the padding, the peaks are eliminated and the SD becomes lower as the results shown in Table 6.

Table 6: Statistical results of 4-thread OpenMP solution on small image with and without padding.

	Time(us)	SD	Speedup	Efficiency
without padding	4.71	4.38	1.4	0.4
with padding	4.77	0.51	1.4	0.4

5.1.3 Key Point 3: Efficiency. We discuss the thread efficiency of our 2-thread and 4-thread OpenMP solutions. For large image, the thread efficiency of 2-thread and 4-thread versions are similar. However, for small image, the thread efficiency of both 2-thread and 4-thread versions decrease, especially on the 4-thread version. This statement results in the overhead of threads' creation becomes non-neglectable for dealing with small image.

5.2 Forward DCT

The results of forward DCT are divided into three stages according to Section2.3.3 and a comparison between libjpeg-turbo's and our solutions is provided. Discussion is made to compare the performance and effectiveness of different methods. Because functions F2 (jpeg_fdct_16x16) and F3 (jpeg_fdct_islow) only differ in input size, we put their results side by side and discuss them together.

5.2.1 Stage 1: OpenMP *on inner loop.* Table 7 and Table 8 show the results of using OpenMP on the inner loop process with small image and large image, respectively. The results are very poor for both images and both functions. There are two possible reasons. Firstly, the amount of data in one MCU is too little to overcome the parallelization overhead caused by distributing data and allocating threads. Secondly, data in one MCU are allocated as an array in the main memory and may cause false-sharing problem when they are distributed into different cores or processors. We try to solve this issue by adding padding to the data, and the results are shown in Table 9. We pad the data according to the 64-byte cache line size, but the results do not improve. Potential reason can be that padding data costs extra time to execute, which is more than the time we save.

Table 7: Using OpenMP on the inner loop with small image

	jpeg_fdct_islow		jpeg_fdct_16x16	
threads	time(ms)	speedup	time(ms)	speedup
serial	7.263		10.893	
2	163.504	0.0444	83.524	0.1304
4	245.058	0.0296	119.195	0.0914
8	377.197	0.0193	172.072	0.0633

Table 8: Using OpenMP on the inner loop with large image

	jpeg_fdct_islow		jpeg_fdct_16x16	
threads	time(ms)	speedup	time(ms)	speedup
serial	795.11		1169.341	
2	42891.141	0.0185	8181.994	0.1429
4	51700.676	0.0154	10518.33	0.1112
8	65584.877	0.0121	13162.8	0.0888

Table 9: DCT 16×16 parallelized by OpenMP with padding.

	jpeg_fdct_islow			
threads	small image		large image	
	time(ms)	speedup	time(ms)	speedup
serial	7.263		10.893	
2	248.467	0.0386	20618.379	0.0386
4	329.225	0.0221	24332.578	0.0327
8	522.979	0.0139	31991.352	0.0249

5.2.2 Stage 2: OpenMP *on outer loop.* After failing to parallelize the inner loop of the process, we turn our attention to parallelizing the outer loop, and the results are in Table 10 and Table 11. This time, although we still get poor results using small image, the results of using large image do show some speedups. Based on the different results we get, we can reasonably tell that the small image does not contain enough data to overcome the parallelization overhead. Another thing we have noticed is the low efficiency given by Table 11. We infer that this is also due to the false-sharing problem, but we do not attempt to solve this issue since padding the whole image requires two to three times the memory space, which is not practical in reality.

Table 10: DCT parallelized by OpenMP from outer loop using small image

	jpeg_fdct_islow		jpeg_fdct_16x16	
threads	time(ms)	speedup	time(ms)	speedup
serial	795.11		1169.341	
2	15.835	0.46	25.253	0.43
4	24.432	0.3	28.402	0.38
8	28.484	0.25	26.677	0.41
16	34.207	0.21	29.686	0.37
32	86.88	0.08	34.459	0.32

5.2.3 Stage 3: CUDA *on outer loop.* Since the large amount of MCUs go through foward DCT independently, using data-level parallelism (CUDA) provides a large potential to speedup the process. The evaluation results are shown in Table 12. Contradict to our expectation, the results of using CUDA are even worse than those using OpenMP. The reason lies in the memory operation. As shown in Table 13, the time cost is actually extremely small if we exclude memory operations that move data to and from the GPU. This suggests that the time spent by the memory operation is the bottleneck of foward DCT on images. Moreover, this cannot be overcome by having larger image input since it has more data for the memory operation to deal with.

5.2.4 Comparison to libjpeg-turbo. We analyze the performance of using SIMD to parallelize the foward DCT operation. The results are shown in Table 14. According to the results, libjpeg-trubo's solution has better speedup with large image then small image, and the reason may be the innegligible overhead of using SIMD instructions. SIMD requires data to be packed inside some specialized registers (XMM registers in SSE2), and data need to be unpacked and moved back to the original registers after the calculation. The data re-arrangement described above costs extra time and limits the speedup to deal with small image. Make a cross comparison to the two data-level parallelisms with Table 12, Table 13 and Table 14, although CUDA has better performance for calculation, the overhead of data re-arrangement in SIMD is lower than CUDA. As the result, for processing the small-scale data such as MCUs, SIMD is more suitable than CUDA. On the other hands, in accordance to Table 11, for large image, OpenMP solution is able to performs better than SIMD. For forward DCT, the processed MCU size and input image size are independent, so the increment of threads creation overhead effects less than workload on each thread while input image size increases; otherwise, the data re-arrangement overhead of SIMD increases linearly when the input image size increases. With a higher effects to performance on workload can benefit the computation ability of each thread.

5.3 Summary

In this section, we summarize the evaluation results, comparisons and discussions above. For data-level parallelism, CUDA and SIMD, data re-arrangement overhead becomes innegligible when processing small-scale data. Considering without data re-arrangement

Table 11: DCT parallelized by `OpenMP` on outer loop using large image

	jpeg_fdct_islow			jpeg_fdct_16x16		
threads	time(ms)	speedup	efficiency	time(ms)	speedup	efficiency
serial	795.11			1169.341		
2	1030.188	0.77	0.39	938.62	1.25	0.62
4	544.671	1.46	0.36	504.914	2.32	0.58
8	284.559	2.79	0.35	302.95	3.86	0.04
16	216.257	3.68	0.23	353.918	3.3	0.21
32	226.743	3.51	0.11	350.327	3.34	0.1

Table 12: DCT parallelized by `CUDA` on outer loop.

	jpeg_fdct_islow				jpeg_fdct_16x16			
threads/block	small	speedup	large	speedup	small	speedup	large	speedup
serial	7.263		795.11		10.893		1169.341	
256	1171.295	0.0062	2302.36	0.35	1193.202	0.01	3451.549	0.34
512	1164.745	0.0062	2467.472	0.32	1222.001	0.089	3437.493	0.34
1024	1177.532	0.0061	2361.799	0.34	1192.433	0.009	3432.811	0.34

Table 13: DCT parallelized by `CUDA` excluding memory allocation and copy on outer loop.

	jpeg_fdct_islow				jpeg_fdct_16x16			
threads/block	small	speedup	large	speedup	small	speedup	large	speedup
serial	7.263		795.11		10.893		1169.341	
256	0.027	269	0.032	24847.2	0.024	453.9	0.029	40322.1
512	0.028	259.4	0.034	23385.6	0.023	473.6	0.021	55682.9
1024	0.033	220.1	0.035	22717.4	0.022	495136.4	0.022	53151.9

Table 14: jpeg_fdct_islow compare with `libjpeg-turbo`

	small image		large image	
threads	time(ms)	speedup	time(ms)	speedup
serial	7.263		795.11	
turbo	3.983	1.82	320.855	2.48

overhead, `CUDA` owns the better performance. But, when considering data re-arrangement overhead, SIMD is a better choice with a stably performance.

For function-level parallelism, `OpenMP`, the overhead comes to the thread creation. The overhead makes the thread efficiency low while processing small-scale images. Otherwise, for large-scale images, the overhead is diluted and makes better efficiency.

For a comprehensive discussion, color-space conversion states a problem with a relation that its workload depends on the length of row of the input image, we called it ẅorkload-input-dependent.̈ To deal with the workload-input-dependent problem, data-level parallelism is more suitable than function-level parallelism that SIMD has stably parallelization results on both small and large images, and `CUDA` has the best result on large images. On the other hand, forward DCT states a problem without workload-input dependency that its workload of each thread depends on the pre-defined size of the MCU and has less dependency of the input image. In this problem, the function-level parallelism (`OpenMP`) suffers less from the overhead of thread creation than SIMD's overhead of data re-arrangement when the input image size increases. Hence, with balanced configurations between the number of threads and the workload on each thread, `OpenMP` is able to have a better performance than SIMD on this problem. As the results, the workload-input dependency is possible to help us with choosing the parallelism technologies for image codec parallelization.

6 CONCLUSIONS

In this project, we select and accelerate three hot spots of `libjpeg`'s image compression process with both function-level parallelism and data-level parallelism. Also, we compare our solutions with `libjpeg-turbo`, which is a popular project based on `libjpeg` and has been used in many applications. `libjpeg-turbo` use SIMD instruction set extensions for acceleration and it belongs to data-level parallelism. For function-level parallelism, we use `OpenMP` for implementation and get a better results for the hot spots in the forward DCT phase of image compression process. And for data-level parallelism, we use `CUDA` for implementation and get a better results for the hot spot in the color-space conversion phase of image compression process with large input image. Due to the design philosophy of SIMD, it has stably performances on both large and small input image. Moreover, we make a comprehensive discussion of our experiments and comes out the point, called workload-input dependency, assistant to choosing between different parallelism technologies for image codec parallelization.

REFERENCES

[1] 2017. https://libjpeg-turbo.org/About/Performance
[2] 2019. https://www.google.com/photos/about/
[3] 2019. https://github.com/LuaDist/libjpeg
[4] 2019. http://www.ijg.org/
[5] 2019. https://github.com/libjpeg-turbo/libjpeg-turbo
[6] 2019. https://libjpeg-turbo.org
[7] 2019. https://www.openmp.org/
[8] 2019. https://computing.llnl.gov/tutorials/openMP/
[9] 2019. https://docs.nvidia.com/cuda/index.html
[10] 2020. https://en.wikipedia.org/wiki/Color_space
[11] 2020. https://en.wikipedia.org/wiki/YCbCr
[12] A. Gheewala, Jih-Kwon Peir, Yen-Kuang Chen, and Konrad Lai. 2002. Estimating multimedia instruction performance based on workload characterization and measurement. 98 – 106. https://doi.org/10.1109/WWC.2002.1226498
[13] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 2004. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* 39, 4 (April 2004), 49–57. https://doi.org/10.1145/989393.989401
[14] G. Langdon, A. Gulati, and E. Seiler. 1992. On the JPEG model for lossless image compression. In *Data Compression Conference, 1992.* 172–180. https://doi.org/10.1109/DCC.1992.227464
[15] C. Loeffler, A. Ligtenberg, and G.s. Moschytz. [n. d.]. Practical fast 1-D DCT algorithms with 11 multiplications. *International Conference on Acoustics, Speech, and Signal Processing,* ([n. d.]). https://doi.org/10.1109/icassp.1989.266596
[16] G. K. Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (Feb 1992), xviii–xxxiv. https://doi.org/10.1109/30.125072