Performance analysis of JPEG compression parallel implementation – cw2 SET10108

Alessio Williams Gava - 2018

Abstract

This report aims at the application of parallelization techniques on a JPEG compression algorithm and further analysis of performance gains. To give a wider overview, different methods have been used: OpenMP, multithreads using futures functionalities, GPU parallelization with OpenCL.

1. Introduction and Background.

Images captured from a camera are initially stored as RAW format in which all the information related to the shot are saved and for a high resolution camera, it can result in large files to be fit into the memory card. This would highly reduce the number of images that the memory could store. Techniques have been applied to decrease the size of images without visibly affecting image quality, but there was a need to create a standardized image compression in order to provide an easier transfer and storage of images and to enable interoperability between different manufactures' equipment.

A committee known as JPEG (Joint Photographic Experts Group) in 1992, proposed two compression standards: a lossy and a lossless. This report focuses on the former.(Wallace, 1992)

The algorithm is composed by:

- Image conversion from RGB to its luma, blue and red component: YCbCr
- Downsampling, reducing average croma color
- FDCT (Discrete Fourier Transform): image components undergo mathematical function
- Quantizer: image components quantized using different table influenced by the required quality of compressed image
- Huffman Encoder: image encoded to use less bits for most common values

2. Initial Analysis

The JPEG compression algorithm has been performed utilizing a program available on GitHub (Kornel) which is based on the code by Rich Geldreich (Geldreich) with some extra enhancements. The program provides multiple steps in which parallelization has not been attempted: read image from memory, decompress, check the compression success, etc.. Below it can be seen a few of the mentioned functions.

Only compression of the image has been taken into consideration.

Initially it has been run the program with a big size .png image from Hubble Telescope: 4072px x 4133px 29.9MB. Times have been recorded to give a general idea of the speed of sequential processing. It has been chosen an image with odd sizes, to make sure that the program would support images that do not fit in the grid of 8x8 pixels.

Initially it has been run with all default options except for the quality that was set to 40%, to not perform a too aggressive reduction, outputting a 925 kb image in average of 713 ms. The image is not different from eye inspection, and errors calculated from application are within the permitted range, so the compression was successful.

HUBBLEimg	Time taken ms
Quality 1	636.92
Quality 25	684.96
Quality 50	715.32
Quality 75	822.72
Quality 100	1399.96

Table 1: **Time taken to perform JPEG compression** on 4072x4133 px image for different quality factors

Afterwards, it has been investigated how the compression quality would influence time of execution. The same image has it been compressed 25 times for different qualities: 0, 25, 50, 75, 100. Next figure shows the result:

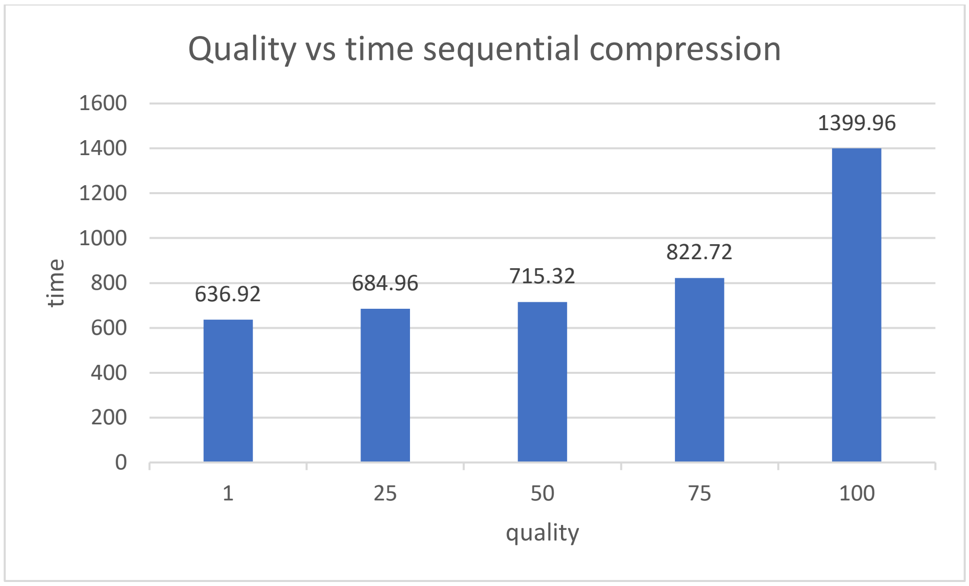


Figure 1: **Time taken to compress large image on different quality settings**

As expected, the low-quality compression took less time. Interestingly, there is not much difference between low and above average qualities, instead on very high qualities there is a high jump on time taken.

It has been noted that utilizing images with different bit depth would take similar time to images having smaller bit depth. It is noticeable in the case of same size images but utilizing RGB – 24 bit depth or RGBA - 32 bit depth. Two images 1280 x 720 with different bit depth have been timed and resulted to take in average 37.88ms and 37.52ms. The extra channel did not influence the algorithm.

Following this result, it has been noted that some images of same size would take quite different time in compressing. For example, it has been used two images both 1920 x 1080 and run both with 20 and 80 difficulty. One of them has many different colors shown, and the other low colors number

	Difficulty 20	Difficulty 80
Image 1 – low n. colors	69 ms	74 ms
Image 2 – high n. colors	92 ms	112 ms

Table 2: **Low number of colors** influencing compression time

As it can be seen, the number of colors in an image will influence the bits needed to use for each color. This has an impact on the subsample and especially on the Huffman encoding which has to count more bits per pixel and do more bits shifting (in general more operations need to be done).

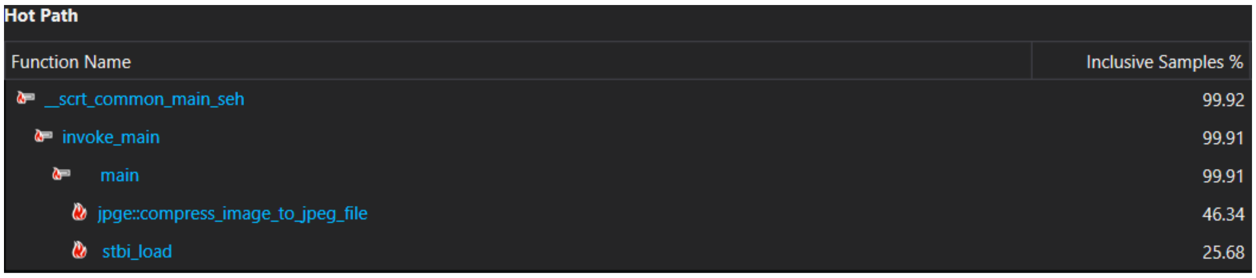
Finally, the size of the image does affect timing of algorithm. It has been tested with different image sizes. The images utilized have four times the total number of pixels of previous size. In order to get a fair environment where it can be checked only image size influence on the time taken, it has been utilized 1 color images (solid color) of different sizes. It has been utilized difficulty of 50.

Size	Total pixels	Time taken (ms)	Ratio with previous image
256x256	65536	2.1	-
512x512	262144	9.4	4.48
1024x1024	1048576	33.3	3.54
2048x2048	4194304	128.4	3.86
4096x4096	16777216	525.3	4.09
8192x8192	67108864	2114.4	4.03

Table 3: **Image size** influencing compression time at difficulty factor 50 with single color

In general, the compression time is proportional of the size of the image provided, in fact, the time increases of a factor of four at each image size.

It has been utilized the Visual Studio integrated Profiler to discover possible areas of heavy CPU use, where it could be attempted parallelization. Below is shown a screenshot of the “hot path” outputted from the profiler.



Hot Path

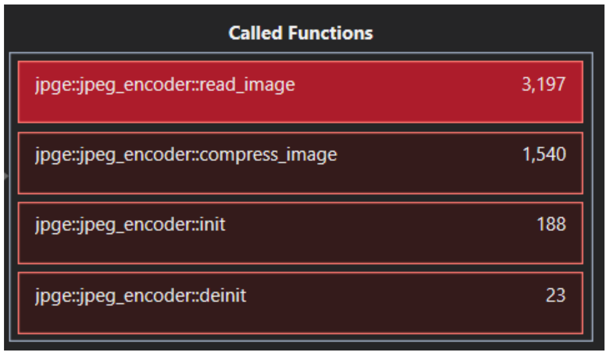
Function Name	Inclusive Samples %
__scrt_common_main_seh	99.92
invoke_main	99.91
main	99.91
jpge::compress_image_to_jpeg_file	46.34
stbi_load	25.68

Figure 2: **Sequential hot path**

The main function comprises most of the work done from the CPU (99.91%), because all the program functionalities are contained in the main(). More relevant are the inner paths compress_image_to_jpeg_file and stbi_load. This information is valuable to start investigating in the provided path for high CPU work done inside those functions. The exclusive sample is 0% meaning that the work is done in inner functions, not directly on the surface of the hot path displayed.

stbi_load is used to load the image into a buffer array, its processes depend on the different type of image provided. In the case of the png, it would decode the Huffman tree contained in the image and then load into an array. Parallel decoding has not been taken into consideration, so stbi_load was not attempted.

It has been explored the compress_image_to_jpeg_file function, leading to the discovery of two main CPU areas of high utilization in which it has been decided to attempt the parallelization: jpeg_encoder::read_image() and jpeg_encoder::compress_image(). The inner operations are suggesting a possible data parallel approach.



Called Functions

jpge::jpeg_encoder::read_image	3,197
jpge::jpeg_encoder::compress_image	1,540
jpge::jpeg_encoder::init	188
jpge::jpeg_encoder::deinit	23

Fig 3: **High CPU consumption areas** in compress_image_to_jpeg_file function

In the functions listed in Fig3 are usual steps in a JPEG compression, in fact:

- read_image() : subsample, load YCC
- compress_image() : dct, quantization, Huffman encoding

3. Methodology

The algorithm has been tested on a machine with specs list on table below:

CPU	Intel Core i7 6700 HQ @ 2.6GHz - x64
Cores	4
Threads	8
OS	Windows 10 Home 64bit
RAM	16GB – DDR4
GPU	Intel HD 530

Table 4: **PC utilized specification**

Following discoveries using the profiler, it has been decided that parallelization will be attempted in the two function read_image and compress_image.

The total time taken is only related to the area parallelized.

There are present multiple nested for loops and in most of the areas there are no dependencies between data. This led to the decision that it would be optimal for data parallel techniques or also called "embarrassing parallel" problems. Initially OpenMP will be attempted to ascertain if a speed up will be possibly achieved. CPU bottlenecks will be discovered using OpenMP, and it will be useful to understand possible areas where there are dependencies between threads. This will provide useful information when successively will be implemented a GPU parallelization, in which no dependencies must be present. It will be used OpenCL framework for programming GPU kernels, providing with information about memory constraint faced by loading buffers into the GPU from CPU environment. Additionally, a manual thread implementation will be created aiming to show overheads present in the managed threads in OpenMP.

No task parallel techniques will be carried out because as mentioned in introduction section, the compression part of the algorithm lends itself to a data parallel application. This is caused from the fact that operations in the compression depend on the total or partial completion of previous processes. For example, Huffman coding needs that the whole image has been transformed, subsample requires that a grid of 2x2 pixels have been transformed in YCC component.

In OpenCL it will be needed to set some barriers because some operations refer to data not modified by themselves, and without a proper control, race conditions would be created.

Huffman coding is not a good example for SIMD processing because of many conditional branches and interdependencies between threads based on values set by other threads. The correct implementation of thread or GPU parallelization would have involved a revolution of the entire Huffman functionality so it has been decided to run it sequentially.

In multithreading, the parts that were kept as sequential, will be executed by one appointed thread only.

To audit the correct functioning of the parallelized application, the image outputted will be checked with the same image produced utilizing same setting in the sequential program. The size of the outputted image has to be the same and it also checked that the statistical error compared to original image matches in the sequential and parallel implementations.

Following the findings on the introduction section, it has been decided to utilize images with same number of colors. It has been utilized .png images of different sizes which were double of previous sizes and using only one color (solid color): 256x256, 512x512x 1024x1024, 2048x2048, 4096x4096, 8192x8192. This ensured a comparable environment between all the images, but at the same time it made the quality factor meaningless because its application would not change any time taken in an image of only one color. For this reason, it has been utilized only quality of 50.

GPU time measurement involved the run of a new program instance every time, because it has been noted that the first run would take longer because of internal GPU setting up (creation of first buffers, getting the platform required). For this reason, it would have been biased towards shorter time measurement if used also the ones from a “warm” GPU.

In OMP, no for loops have been parallelized using the omp functionality: it has been preferred to instantiate the smallest number of threads to not incur on thread creation overheads.

4. Results and Discussion.

The techniques attempted have been timed and their results are proposed below

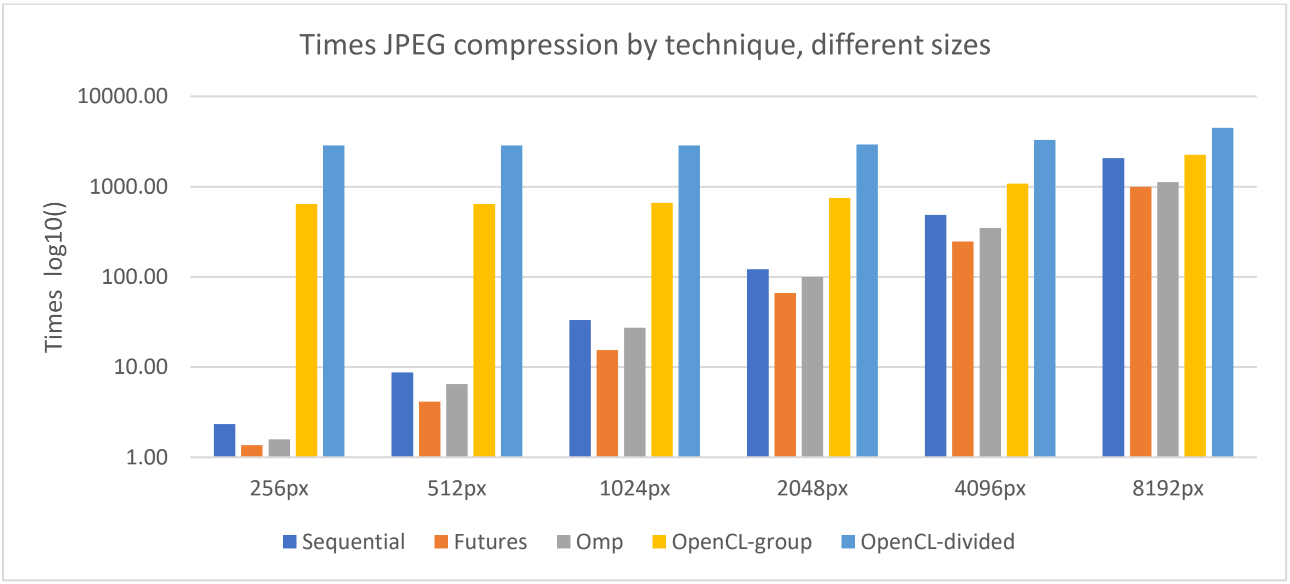


Figure 4**: Logarithmic times taken to perform JPEG compression** by technique for different square image sizes with quality factor of 50

From the figure it can be seen that CPU based attempts, follow an ascending pattern where the time taken increases as the image size increases. As mentioned previously, it increases at a steady rate (factor of four). On the other hand, the GPU attempt do not follow a similar pattern, instead it starts very high for small sized images and it rise of very small percentage for each size of image.

This is because of the fact that CPU approaches do not have much overhead in starting: Futures and Omp have small overhead on the creation of threads, which in total are hardware_concurrency_threads() * 2. The overhead of OpenCL is much higher: building the kernel and copying memory from host to device.

For this reason, two different OpenCL approaches are shown in the figure. The first “group”, initially compiles the kernel files all together, trying to save time for this expensive task. The second “divided”, compiles the kernel files each time they are needed to be run on the GPU.

To show the degree of time consuming, it has been timed a function setting up and running an instance of opencl. It has been considered the call to subsample utilizing OpenCL: total time taken is 300ms, 3 buffers take 5ms each to copy data to device, 10ms to enqueue the kernel and 230ms to build the program. From the data provided it can be understood that the bottleneck in the OpenCL call is the compilation, for this reason a single compilation unit has improved substantially the total time of the application. Below is presented a table with the actual average times.

Pixel square	Sequential	Futures	Omp	OpenCL-group	OpenCL-divided
256px	2.34	1.36	1.58	643.74	2850.00
512px	8.72	4.14	6.48	639.09	2860.00
1024px	33.40	15.50	27.47	663.69	2850.00
2048px	121.50	66.17	99.59	748.59	2920.00
4096px	484.32	247.44	346.79	1080.00	3280.00
8192px	2058.62	1000.00	1120.00	2260.00	4470.00

Table 5: **time taken to execute the application** for each technique and different image sizes in ms

From the times it can be noted that the total time needed for OpenCL to be start up and run its sequential parts: “group” is around 640ms because there is no difference between 256px and 512px meaning that the GPU performs very quickly the image processing and all the other remaining time is because of other preparatory operations. The same issue happens in the “divided”, but as it can be seen the times are higher.

Speedup has been calculated and is presented in the figure below

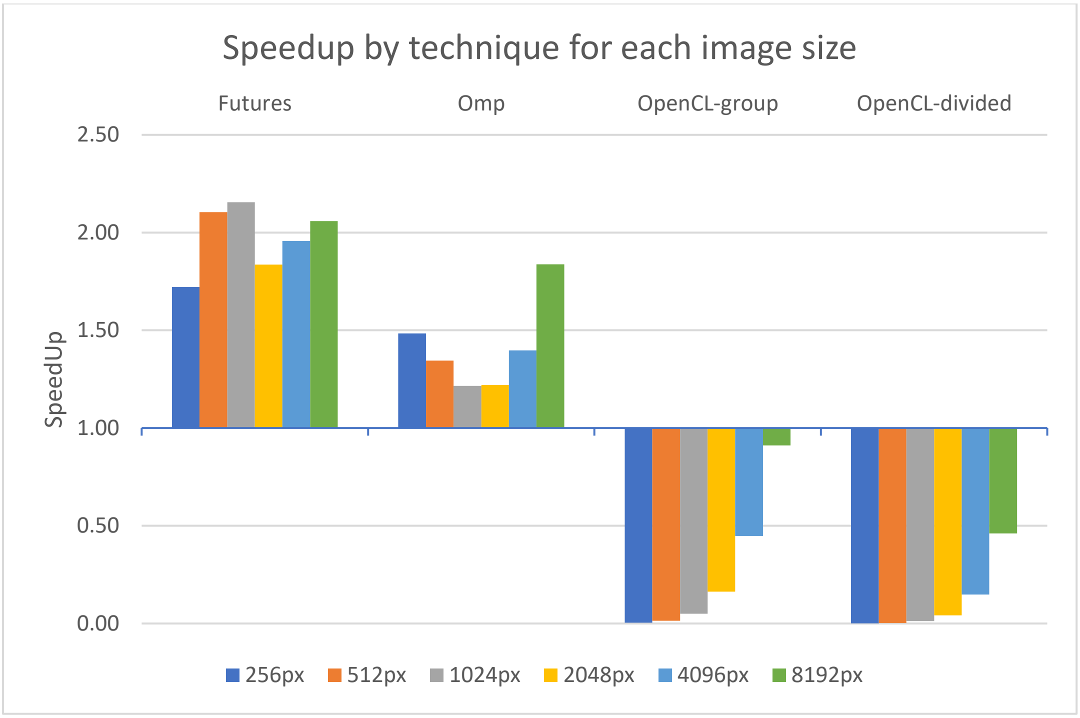


Figure 5: **Speed up** obtained by different techniques for each image size at quality factor of 50

Similarly as Figure 4, the speed up can be divided in two section: CPU based and GPU based.

For the CPU, Futures and Omp performed positively, peaking with 2.15 speed up on 1024x1024 on the Futures technique. In general, Futures are always on top of Omp because it has not being used any form of synchronization compared to Omp where barriers have been put in place to avoid race conditions. Additionally, it has been implemented a functionality to return value from an Omp thread using a vector of booleans. This might be cause of slow down compared to the built in operation of .get() on futures.

In case of small images, it seems that the cost of creating futures is impacting the speed up: 8 threads might be too many for the workload required, in fact bigger sized images perform better. The futures follow a steadier pattern, usually very close to the speed up of 2, instead Omp is more erratic dropping at 1.22 for middle sized images. This might be caused also caused by the utilizing of critical areas in the Omp, resulting in possible situations where a thread would have to wait to pass through it.

Regarding the GPU technique, it can be seen that it was never better than the sequential implementation. For small images the speedup is null: referring to Table 5, the “group” is twice and the “divided” is four time slower. As mentioned, this is because of GPU bottlenecks that do not suit the compression of small sized images. The calculation undergone from the GPU compared to the overhead is very different, resulting in poor performances compared to sequential. Afterwards, the speed up start rising for each image sizes and in “group” from size 4096x4096 the overhead is getting smaller compared to the time taken from sequential. In fact, for the biggest size it almost caught up reaching 0.9 speed up. The overhead on “divided” is still too large to notice any substantial improvement on the speed up.

The efficiency is not easily calculable for GPU environment because it has many cores and not always all of them are running, they are scheduled following GPU directives.

Futures and Omp are utilizing the hardware maximum supported threads, which in my machine is 8. Below is presented a table illustrating efficiency

Pixel Square	Futures	Omp
256px	0.22	0.19
512px	0.26	0.17
1024px	0.27	0.15
2048px	0.23	0.15
4096px	0.24	0.17
8192px	0.26	0.23

Table 6: **Efficiency** for Futures and OpenMP for each image size at quality factor 50

The efficiency reflects the values calculated in the speed up comparing it with the threads utilized. It can be seen that it is quite low (it ranges from 0 to 1), settling around 0.2 which means 20% of ideal improvement has been achieved.

5. Conclusion.

JPEG compression is not a fully parallelizable algorithm because of interdependencies. This creates sequential parts of the code which will limit the maximum speed up gainable. The low performances on Future and OpenMP are affected by this reason. For CPU based techniques, the time taken is highly dependent on the image size because of the limited small number of threads available. On the other hand, GPU showed that improvement could be obtained in case of very large images because of its extensive number of processors. The main overhead encountered for the GPU approach is the compilation kernel time. The issue of slow data transport between host and device, compared to a CPU technique is smaller than in other cases because the program has to use heap memory, reducing difference of transfer speed between GPU techniques and CPU techniques. OpenCL approach is not well suited for small images because the overheads are too much. On the other hand, Futures and OMP demonstrated to deliver improvement even for small sized images.

In general JPEG compression can achieve improvement utilizing parallelization, so if the main focus is in performing faster, it is a viable option. In case the cost of utilizing parallelization is considered, it might not be worth because the maximum efficiency obtained was 27%.

Further improvements will be needed on the GPU application to take advantage of faster local memory between work groups and it could be used precompiled binaries. GPU profiling showed that only 7% of the available power was utilized even when feeding 8192x8192px to it, so investigation on higher GPU usage would be beneficial.

References.

Geldreich, R. *jpeg-compressor*. C++. Retrieved from ttps://github.com/richgel999/jpeg-compressor

Kornel. *Research JPEG encoder.* Retrieved from https://github.com/kornelski/jpeg-compressor

Wallace, G. K. (1992). The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, *38*(1), xviii–xxxiv. https://doi.org/10.1109/30.125072