Concurrent & Parallel Systems - Coursework 2 Report

Glenn Wilkie-Sullivan - 40208762

December 14, 2018

Abstract

This report will comprehensively examine the approaches to parallelising a sequential JPEG compressor, taken from user kornelski on GitHub. These approaches will range from CPU-based parallelism to GPU frameworks and architectures such as CUDA and OpenCL. The ideal result is an optimised JPEG compressor which runs significantly faster with the new approaches implemented.

1 Introduction and Background

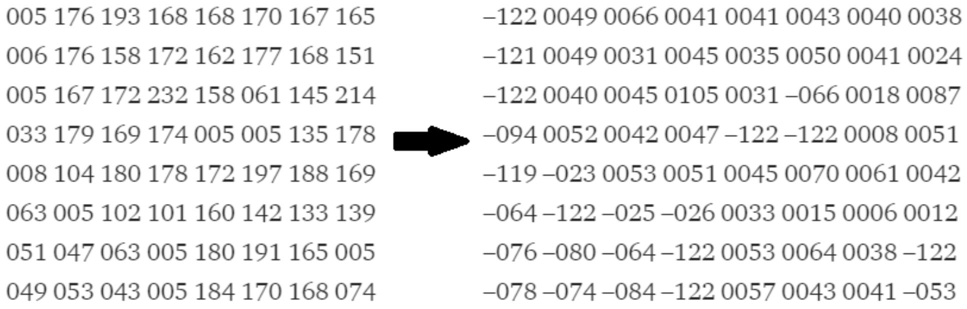
In order to understand how JPEG (Joint Photographic Experts Group) compression can be parallelised, we must first investigate the overall process and purpose of the compression. According to techradar.com, JPEG compression "is a lossy compression format conceived explicitly for making photo files smaller and exploits the imperfect characteristics of our perception". The process for this can be split into five main steps:

- Covert RGB colours of the image to YCbCr (Luminance, Chroma: Blue; Chroma: Red) colour space.
- Preprocess image for DCT (Discrete Cosine Transformation) conversion.
- DCT conversion.
- Coefficient Quantization
- Lossless Encoding

Before analysis is commenced, we must establish an understanding of how these concepts work. In reference to the process list, we will analyse each step of the compression process.

In order to convert RGB colours to a YCbCr colour space, each component transform of the image provides an input value between 0 - 255 and converts it into Y, Cb, Cr values in the range 0 - 255, -128 - 127, and -128 - 127 respectively. The Y (luminance) component is then level-shifted down to the range -128 - 127, and the input tile of the level-shifted symmetric YCbCr colour space is used as the input for the next step of the compression process.

To preprocess for discrete cosine transformation, the image is first split up into equally sized blocks. The pixel matrix for each block is then centralised around 0, and 127 is subtracted from each value. This process looks as follows:



```
005 176 193 168 168 170 167 165      –122 0049 0066 0041 0041 0043 0040 0038
006 176 158 172 162 177 168 151      –121 0049 0031 0045 0035 0050 0041 0024
005 167 172 232 158 061 145 214      –122 0040 0045 0105 0031 –066 0018 0087
033 179 169 174 005 005 135 178  ➡   –094 0052 0042 0047 –122 –122 0008 0051
008 104 180 178 172 197 188 169      –119 –023 0053 0051 0045 0070 0061 0042
063 005 102 101 160 142 133 139      –064 –122 –025 –026 0033 0015 0006 0012
051 047 063 005 180 191 165 005      –076 –080 –064 –122 0053 0064 0038 –122
049 053 043 005 184 170 168 074      –078 –074 –084 –122 0057 0043 0041 –053
```

Figure 1: Discrete Cosine Transformation Preparation

After obtaining these values, the discrete cosine transform (DCT) helps separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). After the image is split into equal blocks, the two-dimensional DCT is computed for each block. The DCT coefficients are then quantized, coded, and transmitted. The JPEG receiver (or JPEG file reader) decodes the quantized DCT coefficients, computes the inverse two-dimensional DCT of each block, and then puts the blocks back together into a single image. For typical images, many of the DCT coefficients have values close to zero. These coefficients can be discarded without seriously affecting the quality of the reconstructed image *(MathWorks, pp. 7, 2018).*

From here, any values left in the pixel matrix close to 0 are converted to 0, then divided by a matrix from the standard JPEG quantisation table (for luminance).

The final step is lossless encoding, using Huffman coding (which the analysed application utilises). This is to preserve the quality of the image while reducing the file size - by using a specific method for choosing the representation for each symbol, this results in a prefix-free code that expresses the most common characters using shorter strings of bits than are used for less common source symbols in the image chunks/parts/spectral sub-bands. No other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code.

We will reference these concepts in the following sections - which involve evaluating the base program, suggesting a better solution for optimisation, implementing it as described, discussing the results and wrapping up the findings. First, the program must be examined and evaluated based on its overall performance.

2 Initial Analysis

For this project, the JPEG compressor created by the user 'kornelski' on GitHub will be analysed. The link for this repository can be found as a reference. The specifications used to run this compressor and analyse it are as follows:

- CPU: Intel Core i7-6700HQ @ 2.60GHz (4 cores, 8 threads)
- GPU: NVIDIA GeForce GTX 960M (4GB, GDDR5)

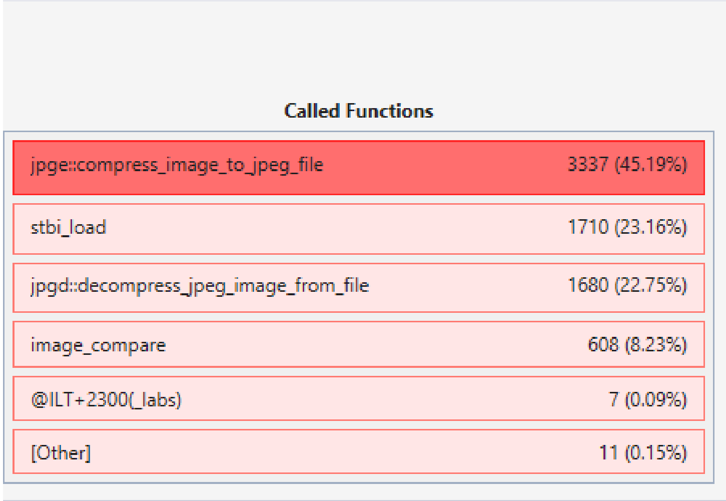
As such, the report will detail a methodology and results assuming these specifications. To start with, I used the Visual Studio 2017 diagnostic tools to analyse the overall CPU usage of the program when given various parameters. The program has a range of functionality, and the most pressing of them is the exhaustive test for the compressor, amongst the general compression and decompresion. For this report, we will look exclusively we at the general compression algorithm, which has multiple variables affecting the execution time. Among these variables is the image size and quality factor. The quality factor is simply a number ranging from 0 - 100, relative to how sharp the resulting image should be. By modifying the quality factor and image size over 20 runs, the following execution times were found:

Image Size	Quality Factor: 40	Quality Factor: 60	Quality Factor: 80	Quality Factor: 100
100 x 100	0.005 seconds	0.006 seconds	0.010 seconds	0.010 seconds
200 x 200	0.0089 seconds	0.0213 seconds	0.0129 seconds	0.0177 seconds
400 x 400	0.0337 seconds	0.0339 seconds	0.0369 seconds	0.0445 seconds
800 x 800	0.1309 seconds	0.1343 seconds	0.1492 seconds	0.1797 seconds
1600 x 1600	0.3461 seconds	0.3586 seconds	0.381 seconds	0.4653 seconds
3000 x 3000	1.0833 seconds	1.0854 seconds	1.1238 seconds	1.2284 seconds

Table 1: JPEG Compression Execution Time

These findings were exclusively within Visual Studio, Release mode, x86. As we can see from table 1, the program itself is very fast. The quality factor of the compression doesn't seem to have much effect on the execution time - in most cases, there was only a rise of roughly 3.5%. When modifying the image size, the rise in execution time can range from between 160% to 288% ± 10% - in order to effectively test the execution time, the image size is the important variable to focus on. When using methods or parallelisation such as multi-threading, OpenMP (Open Multi-Processing) or GPU architectures, it seems likely that at lower image sizes, the initialisation of threads, parallelised loops or quantisation will have more strain on the CPU than the image compression itself. With that in mind, images with a pixel amount larger than 9,000,000 will be far more useful in testing the compression, and will be used as test cases in the following sections. Table 1 will eventually be used as a comparison after a parallelised solution is implemented - for now, the bottleneck(s) of the program must be identified, such that we have a foundational understanding of where the program can be parallelised.

The main hot-path of the program in main can be seen as follows:



Called Functions

jpge::compress_image_to_jpeg_file	3337 (45.19%)
stbi_load	1710 (23.16%)
jpgd::decompress_jpeg_image_from_file	1680 (22.75%)
image_compare	608 (8.23%)
@ILT+2300(_labs)	7 (0.09%)
[Other]	11 (0.15%)

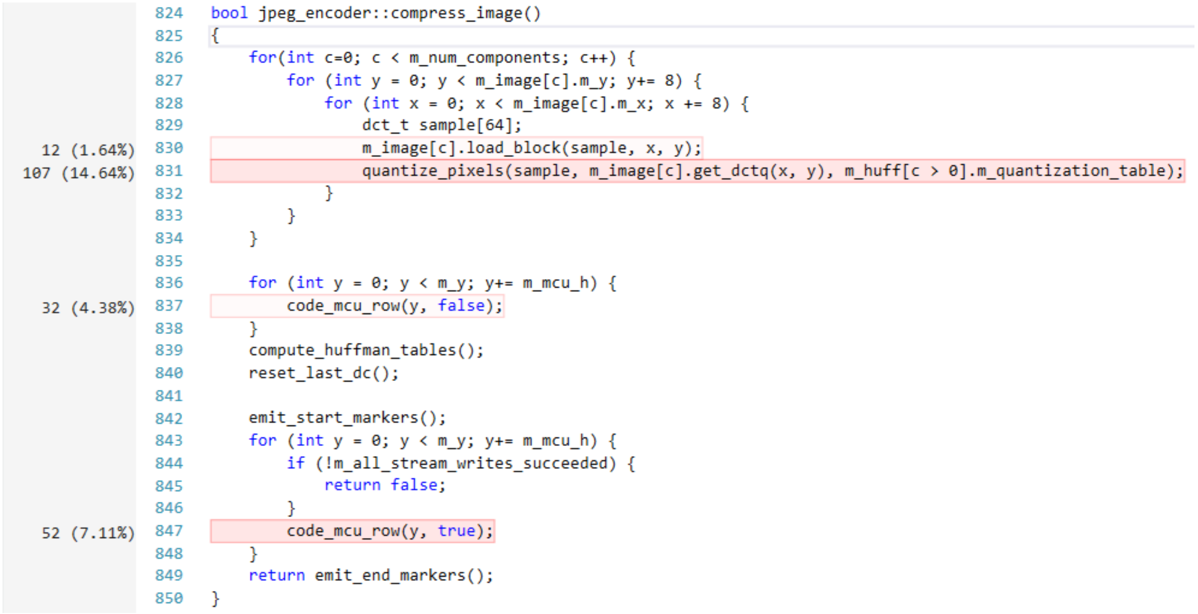
Figure 2: JPEG Compressor Hot-Path

From this, we can extract that the image loading and image comparison functions are costly; these load function simply load the image from a path with predetermined dimension values and parses it accordingly. The image comparison function computes the image error statistics, used to measure the accuracy of the observed image to the expected image. These functions cannot be parallelised, as the operations are inherently sequential. However, the general compression algorithms are pressing the CPU more than these functions; While figure 1 shows the CPU usage within main, we have to look further in the code to find the bottleneck of the program. The almost-full list of function CPU usage is:

Function	CPU Unit Usage	CPU Usage %
jpge::compress_image_to_jpeg_file()	3337	45.19%
jpge::compress_image_to_stream()	3334	45.15%
jpge::jpeg_encoder::read_image()	1930	26.13%
do_png()	1710	23.16%
parse_png_file()	1710	23.16%
stbi_load()	1720	23.16%

Table 2: Individual Function CPU Usage

When looking at the definitions and calling of these functions, it appears as though there are multiple bottlenecks within the 'compress_image_to_jpeg_file' function. The CPU usage for this function is split between reading in the image, taking subsamples of the coordinate data and compressing the image subsequently. A handful of functions from table 2 point to these lines, and it seems logical to assume that this section will be parallelised for better performance using our previously mentioned techniques. When analysing the function itself, the line-by-line breakdown shows some potential areas of parallelisation:



```
              bool jpeg_encoder::compress_image()
              {
                  for(int c=0; c < m_num_components; c++) {
                      for (int y = 0; y < m_image[c].m_y; y+= 8) {
                          for (int x = 0; x < m_image[c].m_x; x += 8) {
                              dct_t sample[64];
  12 (1.64%)                  m_image[c].load_block(sample, x, y);
107 (14.64%)                  quantize_pixels(sample, m_image[c].get_dctq(x, y), m_huff[c > 0].m_quantization_table);
                          }
                      }
                  }

                  for (int y = 0; y < m_y; y+= m_mcu_h) {
  32 (4.38%)          code_mcu_row(y, false);
                  }
                  compute_huffman_tables();
                  reset_last_dc();

                  emit_start_markers();
                  for (int y = 0; y < m_y; y+= m_mcu_h) {
                      if (!m_all_stream_writes_succeeded) {
                          return false;
                      }
  52 (7.11%)          code_mcu_row(y, true);
                  }
                  return emit_end_markers();
              }
```

Figure 3: Potential Spots of Parallelisation

As seen in figure 2, the 'quantize_pixels' function is compressing a range of values related to the image into a single quantum value, such that with less discrete variables, the image should be easier to compress. These discrete values include: sample 'sub-images' which come from the main image, the discrete cosine transform and quantisation of the image dimensions, as well as the values returned from the Huffman[1][2] encoding function. When optimising this application, threaded approaches are generally useful for breaking down and parallelising looped processes, as are parellelised loops supported by OpenMP. As long as the discrete/quantum variables can be shared, the application should benefit greatly from these techniques in the overall speed of execution.

3 Methodology

As seen from the initial analysis, the main bottlenecks lie in the quantisation and image reading processes. There are numerous operations in the program which cannot be parallelised, such as the image I/O functions - as such, we will examine the approaches of parallelising the applicable sections of the program in this section. A selection of parallelisation techniques can be prepared to optimise the application - these techniques can be split into two types: CPU-based, and GPU-based. In this report, we will aim to look at CPU-based approaches such as multithreading, OpenMP parallelisation and MPI, as well as utilising GPU-based approaches such as the OpenCL and CUDA architectures.

The first method of optimisation should and will be general improvements around the application - for such a large program, it seems likely that it contains some superfluous code or functions which could be improved. If there are instances of slow code which can be quickly improved or removed, it will be changed. After this is done, the parallelisation techniques should be implemented and executed in order of difficulty. Starting off with the easiest - OpenMP will be used as a proof of concept that particular facets of the program can be parallelised and optimised. If applicable, the multiple occurences of OpenMP parallelisation should utilise the available integrated functions to make the process more robust. However, proving the compression process is optimised will be difficult - parallelisation can be proved by methods such as OpenMP's built-in functions or printing the thread numbers. However, while the application may be faster, the only way to test the compression is comparing the file sizes. In this report, the findings will reference the execution time of the application and the difference in file size.

Following this, a multithreaded approach will be implemented and tested, as well as a possible MPI solution. Theoretically, in the 'compress_image' and 'read_image' functions where a bottleneck exists, the task could be distributed between a number of threads relative to the CPU; As long as the resources are shared during the quantisation process, then the application is almost guaranteed to be faster. Similarly, if MPI is used, the inbuilt functions can distribute the parallelism among the tasks to improve performance.

[1] Huffman coding is a lossless data compression algorithm.

[2] Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data.

In order to measure the speedup and efficiency, the execution time of each run must be serialised. For unbiased and high-coverage results, the program execution times in the results section will be an average of 20 runs of the application. The speedup and efficiency values will be presented in a table as such (assuming a quality factor of 100, given its relatively low effect on performance):

Image Size (Pixels)	Sequential Time	Parallel Time	Speedup	Efficiency
x pixels, y pixels	w seconds	x seconds	y seconds	z seconds

Table 3: Speedup and Efficiency Results

Where x seconds is the time before implementation, and y seconds is the time after implementation. These tables may have more or less rows depending on the amount of parallelisation for that technique - the time before implementation will be taken from table 1. To visualise these results, R will be used to plot the average or overall time for each technique, for ease of analysis.

4 Results and Discussion

Before any results are presented - the following results were all found within Visual Studio 2017, Release Mode, x86 with the forementioned CPU/GPU specifications. Later in this section, an R script was used to plot the execution times of each technique against the respective image sizes, assuming a quality factor of 100 each time. Firstly, for each technique used, the execution time was serialised into a CSV file, and the average time of 20 runs is taken as the estimate time of execution. As discussed before, a selection of techniques were used to parallelise this program - multithreading, OpenMP, OpenCL and CUDA.

4.1 CPU-based Techniques

Starting with the easiest implementation, OpenMP, this involved introducing parallel loops into the application. The bottlenecked areas lie around the read_image and compress_image functions. With this in mind, multiple parallelised loops were placed in these functions where iteration occurred - the following execution times and speedup was found:

Image Size (Pixels)	Sequential Time	Parallel Time	Speedup	Efficiency
100 x 100	0.0039 seconds	0.0501 seconds	0.0778	1.0006
800 x 800	0.10755 seconds	0.1815 seconds	0.5926	0.99994
1600 x 1600	0.28455 seconds	0.34975 seconds	0.8136	0.99997
3000 x 3000	0.7453 seconds	0.67925 seconds	1.0972	1.00004
8000 x 8000	3.972 seconds	3.1194 seconds	1.2733	1.00002

Table 4: OpenMP Speedup and Efficiency Results

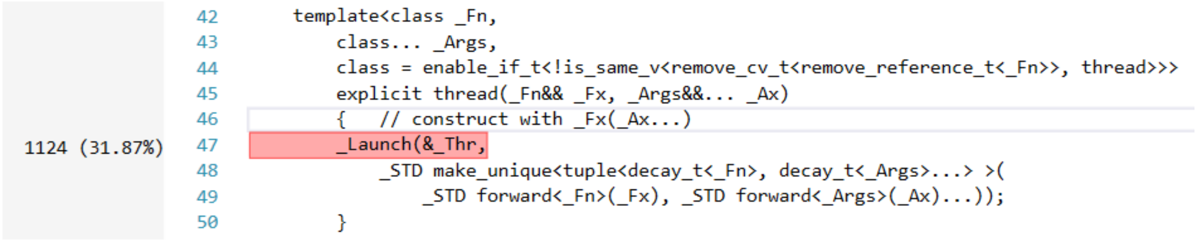
As seen from the above table, the usefulness of OpenMP and it's reliability will only grow exponentially relative to the image size - i.e. as the image gets bigger, OpenMP parallelisation will make the compression process faster. However, at smaller image sizes, the initialisation of parallel loops is more costly than compressing the image sequentially. In the application, we can safely assume that the threshold of parallelised speedup lies at an image size of roughly 2000x2000 pixels. The inclusion of scheduling, whether it be static or dynamic, is completely superfluous - the mapping of threads to relative tasks in the loop cannot speedup the program due to the size of the task. Similarly, resources do not have to be shared amongst the costly functions. In some instances of the program, the variables have to/must be shared amongst loops, such that the output stays correct. However, on examination, this wasn't necessary for any speedup within the program. As a comparison, the multithreading example will be detailed.

As a reminder, in the multithreading example, a CPU with 8 threads will be used as the context of analysis. The implementation of multithreading within the application was very similar to the OpenMP method - for the amount of threads relative to the processor, run a given task among those threads. However, this technique had a significantly slower runtime, as seen in the following results:

Image Size (Pixels)	Sequential Time	Parallel Time	Speedup	Efficiency
100 x 100	0.0039 seconds	0.3627 seconds	0.01075	1.00025
800 x 800	0.10755 seconds	3.1934 seconds	0.03368	0.099996
1600 x 1600	0.28455 seconds	10.4485 seconds	0.02723	1.00013
3000 x 3000	0.7453 seconds	32.44785 seconds	0.02297	0.99996
8000 x 8000	3.972 seconds	223.42575 seconds	0.01778	0.99987

Table 5: Multithreading Speedup and Efficiency Results

The main issue around the threaded approach is the launching/initialisation of threads. As seen in the following figure:



```
                  template<class _Fn,
                      class... _Args,
                      class = enable_if_t<!is_same_v<remove_cv_t<remove_reference_t<_Fn>>, thread>>>
                      explicit thread(_Fn&& _Fx, _Args&&... _Ax)
                      {   // construct with _Fx(_Ax...)
1124 (31.87%)         _Launch(&_Thr,
                          _STD make_unique<tuple<decay_t<_Fn>, decay_t<_Args>...> >(
                              _STD forward<_Fn>(_Fx), _STD forward<_Args>(_Ax)...));
                      }
```

Figure 4: Thread Launching Bottleneck

The launching of a thread, even when compressing an image as small as 1600x1600, quickly becomes the most costly function in the application - using 31% of the CPU resources. Assuming the functions utilising multithreading are being iterated a large amount of times, it makes sense why the launching of so many threads would bottleneck the program. The parallel time, just like the sequential time, seems to scale with the image size, almost exponentially. Even when using a mutex within the forementioned blocks of code, the application is still incredibly slow. Based on these results we can conclusively say that the launching of threads in this context is far more costly than initialising parallel loops within OpenMP. As there are two opposite types of results (both CPU-based), comparing them to another technique should show which of the two is the outlier. The technique to be analysed now will be CUDA, a GPU parallelisation technique.

4.2 GPU-based Techniques

5 Conclusion

References

[1] A. Barnwal. Huffman coding — greedy algo-3. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

[2] D. Dias. Jpeg compression algorithm. https://medium.com/@danojadias/jpeg-compression-algorithm-969af03773da.

[3] kornelski. jpeg-compressor. https://github.com/kornelski/jpeg-compressor.

[4] D. Marshall. The discrete cosine transform (dct). http://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html.

[5] MathWorks. Discrete cosine transform. https://uk.mathworks.com/help/images/discrete-cosine-transform.html.

[6] MaximumCompression. Data compression theory and algorithms. https://www.maximumcompression.com/algoritms.php.

[7] Microsoft. 3.1.8.1.3 color conversion (rgb to ycbcr). https://msdn.microsoft.com/en-us/library/ff635643.aspx.

[8] P. Plus. All you need to know about jpeg compression. https://www.techradar.com/news/computing/all-you-need-to-know-about-jpeg-compression-586268/2.

[9] TechTerms. Lossless definition. https://techterms.com/definition/lossless.