Concurrent & Parallel Systems - Coursework 2 Report

Glenn Wilkie-Sullivan - 40208762

November 24, 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 transformation.
- Coefficient Quantization
- Lossless Encoding

We will touch more on 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 it's 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 100 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% $\pm$ 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:

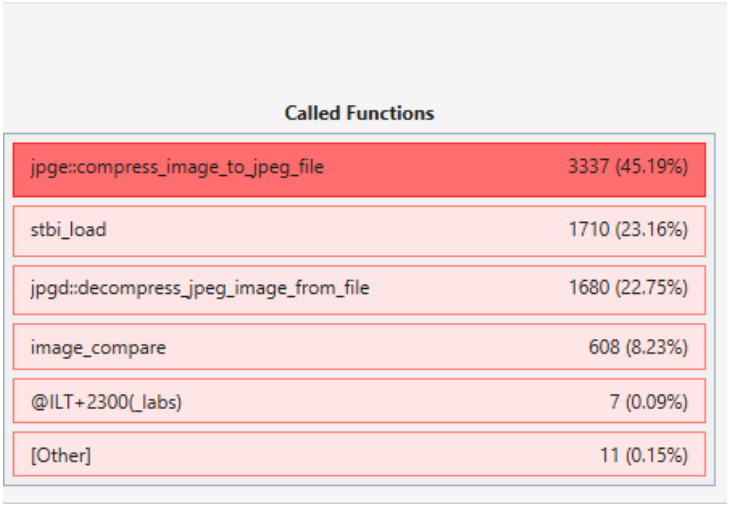

Figure 1: JPEG Compressor Hot-Path

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 co-ordinate 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:

```
                824  bool jpeg_encoder::compress_image()
                825  {
                826      for(int c=0; c < m_num_components; c++) {
                827          for (int y = 0; y < m_image[c].m_y; y+= 8) {
                828              for (int x = 0; x < m_image[c].m_x; x += 8) {
                829                  dct_t sample[64];
     12 (1.64%) 830                  m_image[c].load_block(sample, x, y);
    107 (14.64%) 831                 quantize_pixels(sample, m_image[c].get_dctq(x, y), m_huff[c > 0].m_quantization_table);
                832              }
                833          }
                834      }
                835
                836      for (int y = 0; y < m_y; y+= m_mcu_h) {
     32 (4.38%) 837          code_mcu_row(y, false);
                838      }
                839      compute_huffman_tables();
                840      reset_last_dc();
                841
                842      emit_start_markers();
                843      for (int y = 0; y < m_y; y+= m_mcu_h) {
                844          if (!m_all_stream_writes_succeeded) {
                845              return false;
                846          }
     52 (7.11%) 847          code_mcu_row(y, true);
                848      }
                849      return emit_end_markers();
                850  }
```

Figure 2: 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] 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.

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

3 Methodology

4 Results and Discussion

5 Conclusion

References

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

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

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

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