CONCURRENT AND PARALLEL SYSTEMS

Coursework 2

40173086

Troscianko, Marcel

JPEG File Compression

# Abstract

This project will investigate the JPEG compression algorithm, then attempt to increase it’s performance through the use of multiple parallelisation techniques.

# Introduction and Background

The JPEG file format uses a multi-part lossy compression algorithm and psychological curiosities to reduce an image’s file size while resulting in an impressively subtle drop in visual fidelity. While the general ideas behind this algorithm are well-known and come in many difference flavours, the algorithm itself is surprisingly hard to come by – one can barely spend a moment investigating JPEG before tripping over “Discrete Cosine Transforms” and a dozen ways of explaining them, yet entire days of intensive research into the standard resulted in mere scraps of concrete information and half-working example implementations. But in the end, enough scraps were gathered to put together an impressively functioning solution.

JPEG compression goes through multiple stages of lossy and lossless data compression, which are as follows:

* Converting the pixels to a YCbCr colour space
* Downsampling the image
* Splitting the image into 8x8 blocks
* Performing Discrete Cosine Transforms on each block
* Quantising each block
* Performing Huffman Entropy Encoding on each block
* Saving the blocks as a binary file.

# Initial Analysis

JPEG compression is a complex process consisting of multiple independent operations which tax the system in different ways, so the compression function was split into smaller logical steps to help find potential bottlenecks. These steps are as follows:

* Setup – Empty arrays are created, and the working image is loaded into memory. This step is a lot of sub-steps combined because regardless of the results they couldn’t really be parallelised meaningfully.
* YCbCr – The RGB pixels are transformed to the YCbCr colour space.
* Splitting – The array of all pixels is split into 8x8 pixel blocks.
* DCT – Each block is run through a Discrete Cosine Transform function.
* Quantizing – Each block is quantised, the first major step in file size reduction.
* Huffman – Each block is run through the Huffman Entropy Encoding algorithm.
* Saving – The compressed image is saved to the disk as a functional JPEG file.

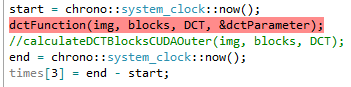
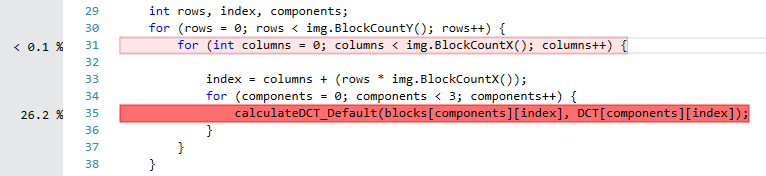
The sections were all timed individually but run consecutively to maintain their combined workload. To find out how the algorithm scales, it was run using four near-identical images of different sizes. Resolutions commonly used in a 4:3 aspect ratio were chosen, with extra care being taken to select ones with a similar scaling from the last. This was done 10 times to ensure correct results, after which anomalous times were discarded and extra tests were run to ensure each result below is a sum of 10.



Figure : Table of results for serial JPEG compression

The first thing that stands out about these results is how long is spent processing DCT transforms – an average of 96.6% of total execution time. The scale of this disparity is hard to picture, so a graph was made to show just how long the DCT times are in comparison to everything else:

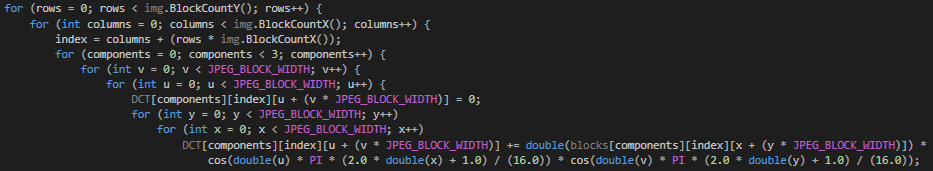
The graph is purposefully left without logarithmic scaling. The results were so disproportionally larger that, in case manual time-keeping had something to do with overall performance around DCT, the test was run again on a 1920x1440 image, this time without manual clock checks. Instead, the Visual Studio Performance Profiler was used.

 The only red-highlighted result was the DCT function. Taking a closer look at it revealed two areas of interest where the execution time appears to have been excessive:

Firstly, an oversight in the coding of the array meant a variable was recreated every turn on the loop, which was also the only loop counter variable to have any impact on performance. More importantly, the execution spent on the DCT calculations appear to have been so high the only other “intensive” line took up less than 0.1% of the checks. The Profiler claims the most intensive part of the program only took up 26.2% of it’s total checks, but that’s because another program had been running in the background taking up and extra 73.5%:



Clearly, the bottleneck of this application are the DCT calculations. Taking a look at the code instantly shows why this is the case:



The DCT calculation itself is nested within a 7-level for loop, giving this program a Big O complexity of O(N7). This likely means the operation itself isn’t particularly demanding, rather the high-nested nature of it alone causes the slowdown. It was decided that the DCT calculation function would be the focus of this parallelisation effort as the potential speedup in reducing it would be much higher than that of any of the other 3.4% processing functions.

The nature of a deeply nested root makes parallelisation trickier – the outermost layer may not contribute significantly to the loop calculation time, while nesting the overhead of CPU threads or GPU memory pushes deep in the loop is likely to further harm performance.

# Methodology

## General Approach

All tests were conducted using the same variables to guarantee correctness of results. Each approach was tested using the same image scaled to four resolutions:

1024x768, 1280x960, 1600x1200, and 1920x1440.

The 4:3 aspect ratio was chosen because JPEG compression divides images into 8x8 blocks, and 4:3 allows smaller files that are still divisible by 8. The resolutions were picked because their size increases in steps of 1.5x, allowing results to be meaningfully compared.

The image selected was scaled down from 1920x1440 to the other resolutions as this means information is lost, not generated. Scaling up would generate information causing a blurrier image – pixel similarity reduces the processing load on the JPEG compressor with each step.

Each test was taken twice – once just to get the CPU, GPU and Memory usage during each approach, and again to time each component of the compressor and see how these differ with approaches. This was done because the Initial Analysis showed most of the components were fast to process, so the mere act of timing them could skew the results. Both tests were ran 10 times to ensure the output hasn’t been unfairly skewed by background processes.

The components tested independently were:

Image loading, RBG->YCbCr conversion, Block splitting, Applying Discrete Cosine Transforms, Quantisation, Huffman encoding, and File saving.

Both OpenMP will be run on 8 threads as a “main” approach, and this approach is what the charts will be based on. The extra threads investigation with OpenMP *static* schedule will be analysed in respect to other *static* results.

## Technologies Used

### OpenMP

Open Multiprocessing is a library that abstracts the creation and management of CPU threads and thread-bound variables. The ideology behind it is virtually the same in every language it’s written for, but in C++ specifically it works using #pragma pre-processor directives. The programmer must simply use place one above loop, define the amount of threads, scheduling, and how to treat the variables, and the library takes care of the rest.

#### Why?

The largest chunk of processing in JPEG compression is as a result of DCT transforms. Normally, large number-crunching tasks would be best suited for GPU processing, however:

* The DTC function is within a seven-layer nested loop and doesn’t take much power to process on its own – the actual maths alone isn’t intense enough to use a kernel for. Memory transfer times between the CPU and GPU could potentially eliminate any speedup gained.
* The DTC function reads and writes to a 3D array/vector – this isn’t something the GPU is built to deal with very well. Flattening these arrays then expanding them after processing could is a very involved task on the side of the programmer.

In short: Using GPU kernels deep inside the nested loop will strain memory without advantage of the processing power, while placing them higher up will be very challenging to implement. This means makes CPU multithreading the simplest to implement and most guaranteed way of increasing performance. CPU multithreading is often done with plain C++ threads but in this scenario, it’d mean on-the-fly creation and destruction of threads in a round-robin fashion imitating a thread pool. OpenMP’s use of a real thread pool avoids the issue entirely thus minimising overhead.

OpenMP comes in multiple flavours referred to as *schedules*, which determine how OpenMP splits the workload between its threads. There are four different schedules to choose from; *static*, *dynamic*, *guided*, and *runtime*.

#### Static Scheduling

The default OpenMP setting, *static* scheduling splits the workload into equal-sized blocks of a user-determined size, then performs its task on them. Upon completing the tasks on all its block, a *static* thread will wait until all other blocks in its current iteration have finished to. OpenMP will then split the next chunk of the workload into blocks, and the cycle continues. This ensures blocks are processed at roughly the same time and in order.

The strength of this lies in uniform workloads – if blocks are very similar in size then the threads won’t be idle very often, while the splitting of blocks and processing itself is faster as *static* scheduling includes much less of an overhead than the other types.

#### Dynamic Scheduling

Though very similar, *dynamic* scheduling can in many ways be thought of as the opposite of *static* scheduling. It too splits the workload between threads by a pre-set chunk size. Unlike in *static* scheduling however, each thread is then free to process chunks until all the work is done. A thread that completes its work will automatically ask for more in a round-robin fashion, resulting in out-of-order processing. This means varied-size workloads are processed much faster in *dynamic* than *static* scheduling because the threads aren’t bound together by a synchroniser.

The constant thread-to-dataset calls aren’t free however, and result in a larger overhead than in *dynamic* scheduling. This overhead is often enough to make it run slower on uniform workloads than *static* scheduling would.

### CUDA

Nvidia’s own take on GPU parallelisation, *CUDA* allows the programmer to program GPU kernel code without the need the for the graphical shaders used in languages like OpenGL or even separate kernel files, making it ideal for General Purpose Computing on Graphics Processing Units (*GPGPU*). Instead, the developer writes very C-like code straight into their *main* class and can reference it from within regular CPU code. While so far it sounds much like *OpenCL*, *CUDA* is generally understood to run similar code noticeably faster as well as having full support of dual-GPU setups.

Being a GPU-based approach gives *CUDA* programming an huge upper hand when utilised properly, as it’s multi-hundred-core structure allows even entire images to be processed in a few passes. The cost of this is per-processor efficiency – giving a single GPU core a large task to process will result in far slower performance than on a CPU core.

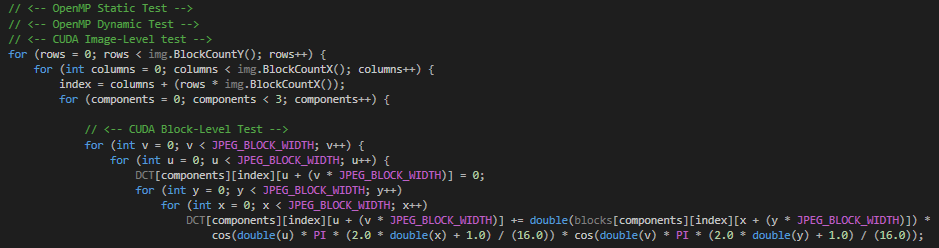
Another drawback is the physical location of the GPU – writing to it isn’t nearly as quick as writing to cache on a CPU and the same is true for reading. It is generally recommended to allocate substantial amounts of memory on a GPU before using its kernels to increase efficiency, and to split a large task between many GPU cores in one go. Splitting a task on the CPU first will result in unnecessary GPU memory writes which can cripple performance.

This is also the problem with the task at hand. Running the *CUDA* kernel from anywhere within the loop will result in many writes and reads from the GPU. On the other hand, placing the kernel code outside the loop will require a quintuple-nested loop be executed on each GPU core. Besides there being many repeats of the same operation, however, the loop isn’t very calculation-heavy, so it may not affect the GPU cores at all. Both the memory-write-intensive and the core-execution-intensive operations will be tested to settle this.

## Test Plan

The problem being solved is a 7-level nested for loop, so any overhead will be consequential regardless of size. While *static* scheduling could do well a few levels into the loop, *dynamic* scheduling’s overhead would make it ineffective and *CUDA*’s need to write to an external memory on every iteration is likely slow the function down more than speed it up. For a fair comparison, the three tests will be conducted on the outermost loop (on a per-image level),

*0 <= rows < blockCountX.*

Memory writes also aren’t a thing usually taken into consideration when optimising. To test how this could affect a program, another *CUDA* kernel will be run within the 3rd-level of nesting. This is the level at which the pixel arrays are split into 8x8 blocks, so the data size processed by the block-level kernel will be far smaller, but it will be re-run many consecutive times. This test will hopefully show the processing-time vs. writing-time exchange that comes with GPU processing. 

Though right now it seems like blocks aren’t being split off, the code will be altered to send go into a subroutine where the last *CUDA* test takes place, sending single blocks as parameters like so: 

# Results and Discussion

Every shown attempted test has been shown as a grid of times in milliseconds, showing how the increasing image size as well as each step affected the processing time. Along these times are the Speedup and Efficiency calculated based on the total runtime of each image compression compared to the total runtimes gathered in Section 2: Initial Analysis. The formula used for Speedup is:

Where *T* = the total parallel time, and *Ts* = the total sequential time. The formula used for Efficiency is:

Where *S* is the previously calculated speedup, and *p* is the number of cores in the device used. In CPU-based tasks this was 8 as this was the number of cores on the CPU used. In GPU tasks, this was 2048 as this is the number of *CUDA* cores on the graphics card used. The reason this number is uses instead of the number cores actively used is twofold:

1. Efficiency is the measure of how successfully (*efficiently*) an algorithm uses it’s resources. An algorithm not using all 2048 cores could have been written more *efficiently*, which this approach quantifies.
2. The number of *CUDA* cores being used differs with the size of image used in the algorithm. This means using that as *p* doesn’t give a number that really means a whole lot in relation to the algorithm.

## OpenMP – Static

OpenMP’s *static* schedule test was repeated on different numbers of threads. Doing this only for both OpenMP algorithms wouldn’t tell us much more than testing it on one.

### OpenMP – Static – 2 Threads

### 

### OpenMP – Static – 4 Threads

### 

### OpenMP – Static – 8 Threads

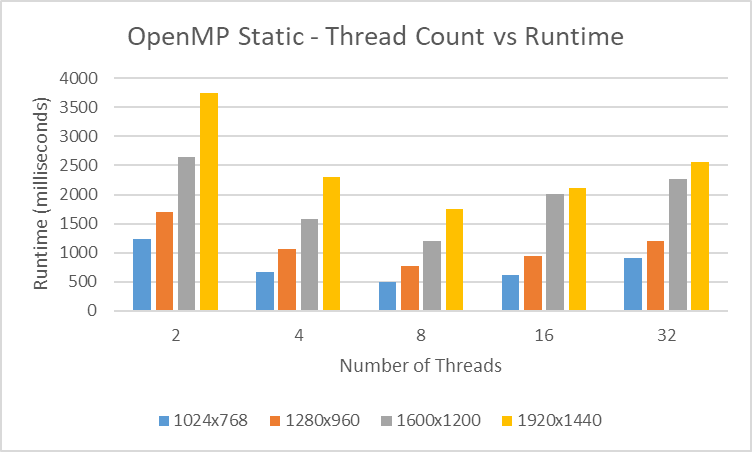
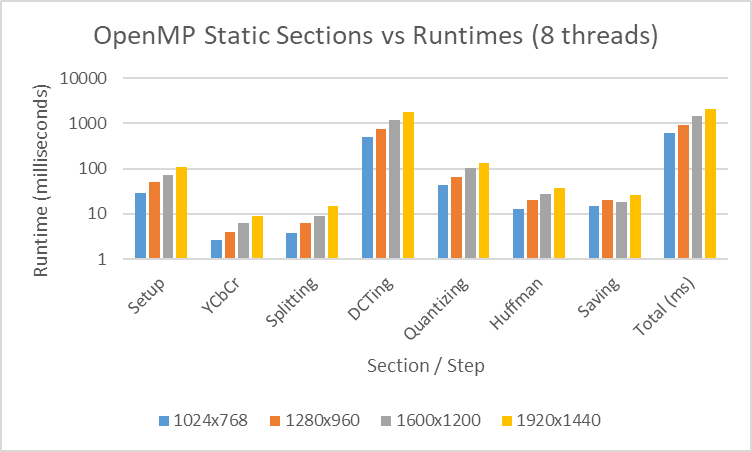
### 

### OpenMP – Static – 16 Threads

### 

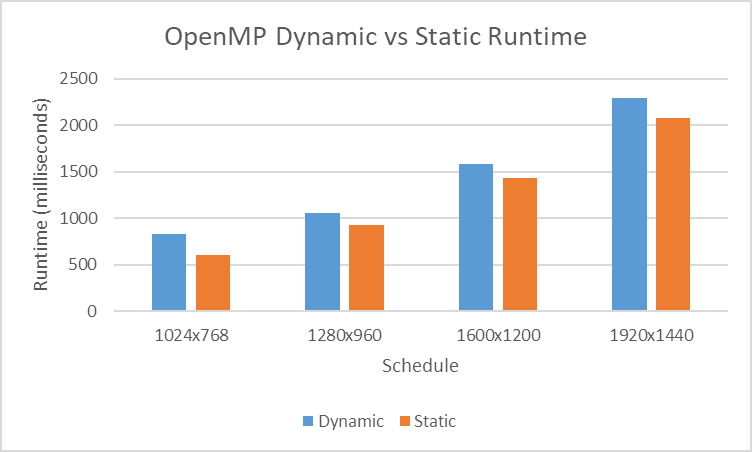
### OpenMP – Static – 32 Threads





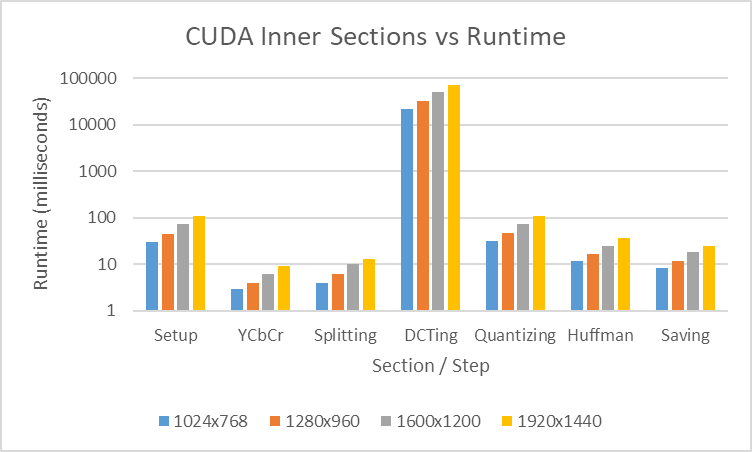
## OpenMP – Dynamic





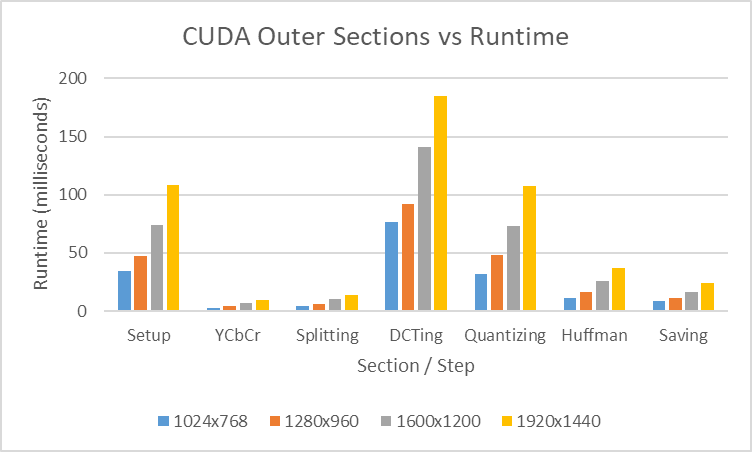
## CUDA – Per-Image





## CUDA – Per-Block





# Conclusion

The goal of the project was to develop a C++ JPEG encoder and then attempt to improve its efficiency. This was done through the use of multiple OpenMP static schedule settings, an OpenMP dynamic schedule, and two CUDA kernel processors.

Start off with OpenMP static, the multiple thread settings seemed to have really made a difference to processing time, causing near 100% variations in Speedup. The tables and graph clearly show that Runtime dips (and so speed and efficiency peak) at around 8 threads. More than that and it starts to slow down, the same is true of any less. This indicates that the optimum number of threads to use on the machine tested is equal to the number of cores on it. While this sounds obvious, testing from a few years back showed about 1.5x the number of cores to be the optimal amount for threading.

The comparison graph between OpenMP Dynamic and Static schedule doesn’t come as much of a surprise. It shows that, on the optimum thread count, Static scheduling beats Dynamic scheduling when it comes to DCT transformation. It is important to note that DCT transforms always conduct the same amount of calculations per block, making it an extreme uniform workload to process. This should be tested on Huffman compression – while it doesn’t require any speedup, it’d be interesting to see how they handle something so data-uniform yet different in execution.

Though it was challenging to get working, I’m glad the CUDA test were run. Initially only the first one worked so I suspected that memory transfer might be what’s making the Inner method so slow. This was ultimately confirmed when the Outer results came back – the Inner CUDA loop slowed the project down as predicted, but to a completely unforeseen level. The outer CUDA, however, sped it up to the same degree. This really shows a developer must understand the tools their using, because otherwise the best techniques can really damage the final product.

While not directly linked to parallelisation, having a fully functioning sub-sampling implementation would be beneficial to determining speed and efficiency changes between approaches. The DCT algorithm the testing focused on would have much less work to do and it’s possible a different approach would come out on top were sub-sampling taken into consideration.