Concurrency and Parallelism Project Report
Iterative Histogram Equalization Parallelization

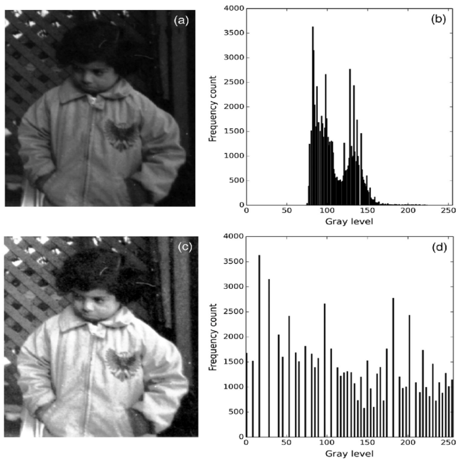
Gonçalo Virgínia
g.virginia@campus.fct.unl.pt

ABSTRACT

The present paper provides a summary of the Concurrency and Parallelism Project of 23/24, which deals with the analysis and implementation of parallel versions of the **Iterative Histogram Equalization** algorithm.

Standard **Histogram Equalization** is a technique used in image processing to enhance the contrast of an image by redistributing pixel intensities. It works by transforming the intensity values of an image so that the histogram of the output image is approximately uniform. This means that the intensities of the pixels are spread out over the entire range of possible values, making the image more visually appealing.
The process of histogram equalization involves the following steps:

(1) Compute the histogram of the input image, which represents the distribution of pixel intensities.
(2) Compute the cumulative distribution function (CDF) of the histogram.
(3) Normalize the CDF to scale the intensities to the range [0, 255].
(4) Map each pixel intensity of the input image to its corresponding normalized intensity using the CDF.
(5) Generate the equalized image using the mapped intensities.



[1]

Iterative Histogram Equalization improves on the standard approach via repeatedly computing the histogram equalization until some convergence criteria is met (e.g. diminishing returns, or pre-defined iterations), which usually leads to much more balanced results, without over-amplifying noise present in the image.

The project is separated into 2 main parts: standard CPU based parallelization - implemented in C++, using OpenMP - as well as GPU based parallelization - using CUDA. With each part delving into the specifics of different choices for each step's optimization.

1 PARALLELISM CANDIDATES ANALYSIS

After observing the provided sequential implementation, several loops - which take on logically different steps in the transformation, as described in the Abstract section - were immediately separated into their own functions, in order to provide a more extensible approach, for both analysis, and the future CUDA version.

The following subsections refer to each individual step, as per their function definition in the actual implementation, and will delve into the individual loop dependencies (or lack there of), alongside any comparisons between applying certain parallel patterns, and any other relevant information that led to the final implementation.

1.1 percentageTo255

As the name suggests, this first loop is essentially just a linear transformation of RGB value representation, from the [0.0, 1.0] float range, to the more typical [0, 255] 8-bit integer values.
Seeing as this first step contains no loop dependencies between iterations, a standard parallel for is suited this step.

1.2 grayScale

Similarly to the previous step, calculating the gray-scale values for each pixel (from 3 channels to 1) does not involve any loop dependencies, and therefore, a standard parallel for is suited for this step as well.

Although, seeing as each iteration accesses 3 sequential positions (RGB) at a time, the thought of partitioning the array into blocks for better CPU cache utilization (much like block matrix multiplication) popped into my mind, although, the experiment did not seem to provide any benefit.

1.3 computeHistogram

The histogram computation is a prime example of a reduction algorithm. In this case, from a specified Height x Width single channel (gray-scale) array of [0, 255] intensities, to an 256-long array of occurrences for each intensity.

This reduction is one of the biggest bottlenecks in the algorithm, since, a parallel reduction of an image - with thousands, to millions, of pixels intensities - into 256 indices, originates an extreme amount of contention between each thread wanting to access the same indices inside the resulting array.
Depending on the intensity distribution and amount of threads, the regular parallel version could even be *slower* than the original sequential version, given the overhead from locking, unlocking, and waiting, between threads, for access to indices in the resulting array.

Knowing this, the solution to this problem is simple: apply OpenMP's reduction(+:histogram) clause to the parallel for.
With this, the original array is split up between threads as usual, but, now each thread instantiates a local/private/intermediate histogram array, for its own mini-reductions, completely avoiding the collisions between threads. At the end, the reduction combines each thread's private histogram into a resulting one, which is no problem even on a single thread.

1.4 computeCDF

On first glance, computing the CDF array didn't seem parallelizable at all, seeing as every iteration contains a loop-carried dependence to the previous one, even more so given the fact that a *prob* function, and minimum calculation were mixed in.

Not much after, it clicked that this was essentially an Inclusive Scan problem, with some extra steps. Although, given the size of the array (256), I doubted it would offer any improvement to the performance, seeing as the overhead would likely balance out any performance gain for such a small amount of iterations. After implementing an Inclusive Scan manually, and another version using the confusing OpenMP Scan related clauses - my hunch turned out correct, performance stayed approximately the same.

In the end, even though the parallel performance gain is relatively small, the *prob* function can be separated from the rest and done first, in parallel, seeing as it does not contain any loop-carried dependencies. From my experiments, a static schedule with 64 elements per thread seemed to provide the best balance given the overhead.
Following this, the loop with dependencies to previous iterations is done, by summing to the previously calculated values, and updating the *cdf_min* variable.

1.5 computeOutputImage

For this last step, the 2 final loops were simplified into 1, alongside the removal of an irrelevant middle conversion to and from [0, 255] in the *correct_color* function.
Following this, seeing as there is an absence of loop-carried dependencies (only containing concurrent reads from the cdf array), a simple parallel for suffices for much performance gain.

2 PERFORMANCE COMPARISON

The following subsections provide a performance breakdown for each function in each version of the implementation.

The resulting values for each function were obtained using the sum of of their running time (in milliseconds, via the *std::chrono* library) over 10 iterations.
The images used can be obtained from the *dataset* folder.

Hardware used:

- CPU: Intel Core i5-8300H (4-cores/8-threads, 4.0GHz, cache: L1: 256 KiB L2: 1024 KiB L3: 8 MiB)
- RAM: 8GB DDR4
- GPU: NVIDIA GeForce GTX 1050 Mobile (4GB VRAM)

2.1 Sequential

As expected, the sequential version has relatively poor performance, when compared to the rest.
Although, we can indeed verify that *computeCDF*'s impact is negligible.

	Image Resolution		
	800x600	3840x2160	5184x3456
percentageTo255	100	1821	3903
grayScale	120	2123	4548
computeHistogram	30	550	1188
computeCDF	0	0	0
computeOutputImage	396	6997	14973
Total	646	11491	24612

2.2 Parallel

From applying just the parallelization techniques described previously, we get a practically 7x speedup when compared to the sequential version.

- Speedup = 24612 / 3970 = 6.2
- Efficiency = 6.2 / 8 = 0.775

	Image Resolution		
	800x600	3840x2160	5184x3456
percentageTo255	18	345	802
grayScale	20	429	994
computeHistogram	0	101	252
computeCDF	0	0	0
computeOutputImage	44	813	1922
Total	82	1688	3970

2.3 Parallel + Vectorization

From applying the vectorization flags on top of the parallel implementation, we get another 7x speedup boost.

Unexpectedly, for the smaller image, each function gave less than 1ms of total runtime over the 10 iterations, so I simply used the general run-time for the total, which still demonstrates a massive speedup.

- Speedup (sequential) = 24612 / 591 = 41.65
- Efficiency (sequential) = 41.65 / 8 = 5.2
- Speedup (parallel) = 3970 / 591 = 6.72
- Efficiency (parallel) = 6.72 / 8 = 0.84

	Image Resolution		
	800x600	3840x2160	5184x3456
percentageTo255	-	81	190
grayScale	-	31	72
computeHistogram	-	3	19
computeCDF	-	0	0
computeOutputImage	-	140	310
Total	11	255	591

2.4 CUDA

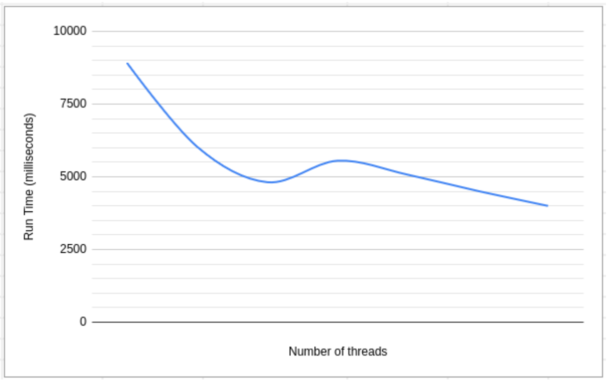
As expected, passing most of the operations to the GPU gave an even greater speedup - although, unexpectedly, the individual functions had a runtime below 1ms on all 10 iterations, most likely because of some implementation error regarding the timer. Given this fact, I used the total runtime as the reference point.
Funnily enough, for the smallest image, the CUDA version performed worse than the previous versions, although that's to be expected, given that passing data to and from the GPU is a well-known bottleneck.

- Speedup (sequential) = 24612 / 224 = 109.88
- Efficiency (sequential) = 109.88 / 8 = 13.73
- Speedup (parallel) = 3970 / 224 = 17.72
- Efficiency (parallel) = 17.72 / 8 = 2.22
- Speedup (parallel-vectorized) = 591 / 224 = 2.64
- Efficiency (parallel-vectorized) = 2.64 / 8 = 0.33

	Image Resolution		
	800x600	3840x2160	5184x3456
percentageTo255	-	-	-
grayScale	-	-	-
computeHistogram	-	-	-
computeCDF	-	-	-
computeOutputImage	-	-	-
Total	63	135	224

3 SCALABILITY

As for studying the scalability in accordance to the increase in workers/threads, I performed multiple runs of the normal Parallel version, with a thread count ranging from 2 to 8, and, as expected, the chart below shows some amount of "diminishing returns". Although, strangely enough, the run time did not decrease as originally thought - for some odd reason, even after running it multiple times, there seems to be some manner of bottleneck in the 5 to 6 thread region, causing a performance bump, only gaining performance with 7 and 8 threads.



4 CONCLUSION

The project was an interesting test-bed for experimenting with different parallelization methods, and, as expected, each step provided an astounding increase in performance, even more than I initially expected, with speedups of practically 7x for the normal parallel version, and another 7x for the parallel-vectored version.
I expected the GPU version to be a bit more prefermant, although, my implementation most definitely didn't squeeze every bit of performance that was possible, especially seeing as there is an actual bug which turns the output to black at some step in the process.

REFERENCES

[1] Alexander Toet and Tirui Wu. 2014. Efficient contrast enhancement through log-power histogram modification. *Journal of Electronic Imaging* 23 (12 2014), 063017. https://doi.org/10.1117/1.JEI.23.6.063017