CP Project - Iterative Histogram Equalization

David Mira, 55906
Diogo Pinto, 67535
d.mira@campus.fct.unl.pt
dmm.pinto@campus.fct.unl.pt

ABSTRACT

Code parallelization is a powerful technique to increase the efficiency of a program. The main benefits of this coding strategy is that the workload is divided among the multiple available threads of the system, making short work of simple yet time-consuming operations. Although parallelization is not the only available coding technique to increase the efficiency of a program it has by far the most potential. In this specific report we analyze the impact that code parallelization has on the efficiency of a image processing program.

KEYWORDS

Parallel processing, Vectorization, CUDA, OpenMP, Parallel Programming

1 INITIAL PROFILING

In the beginning we ran the Clion's default profiler and looked at the code to analyze for hot-spots[2]. We arrived at the conclusion that within the `histogram_equalization` function there were many for-loops that could be optimized. We started by measuring how much time was spent in each loop with the assistance of the "std::chrono" library and printing the results into the console. We tested with a single iteration and with multiple iterations of the image `sample_5184x3456.ppm`[5].

2 PARALLELIZATION CANDIDATES:

After the first analysis of the performance of the program we decide the following loops could very easily be optimized, while significantly improving performance:

(1) **The loop that transforms the image data into unsigned char values**;
Because this loop runs the whole image size, pixel by pixel and performs a simple calculation, which means it can easily be parallelized with great gain in performance.

(2) **The loop that turns the image into gray-scale**;
This loop was parallelized by subdividing the image's row of pixels and assigning them to a thread, this makes the calculation much faster.

(3) **The loop that corrects the color of the image**;
This loop was performing a simple calculation on the whole image so the work could easily be divided into multiple threads.

(4) **The loop that saves the output**;
This loop copies the whole image into the output memory slots, so it means we can subdivide it into multiple threads and accelerate the process considerably.

After the first analysis and implementation of the aforementioned optimizations, we made a second pass through the code, and realised there was one more loop we could optimize, (5) the loop that calculates the histogram of the gray-scale image. Because this loop iterates over the whole image and simply increases an array in a certain index it can easily be parallelized[4].

3 EXPERIMENTAL RESULTS:

We only tested with the `sample_5184x3456.ppm` image file because this was the biggest file, and therefore the file where the differences in processing speed would be most noticeable[1].

It is also worth noting that the CUDA[3] tests could not be run on our machines due to the fact that, neither had a CUDA compatible GPU.

(1) First Tests:

Mode	It = 1	It = 100	It = 500	It = 1000
Iterative	1.97s	~2.5min	>5min	>15min

(2) After Optimizations:

Mode	It = 1	It = 100	It = 500	It = 1000
Iterative	1.97s	~2.5min	>5min	>15min
OpenMP	0.25s	12.08s	58.83s	77.66s

We can clearly see a huge improvement in performance especially when we run the program with a higher iteration count.

We can then take this data and calculate how much of a speedup we get, as well as the efficiency of our optimization (and as a consequence the scalability):

Speedup Formula:

$$S = \frac{T_{seq}}{T_{par}} \tag{1}$$

Efficiency Formula:

$$E = \frac{S}{P}, wherePisnumberofprocessorsused \tag{2}$$

Processor used: Ryzen 5800x3D

This table shows the speedup, and efficiency values of the program, assuming 16 CPU threads:

CycleId	Iterative	OpenMP	Speedup	Efficiency
uchar calc.	0.22s	0.026s	8.46x	52.8%
`gray_img`	0.29s	0.043s	6.74x	42.1%
histogram	0.07s	0.048s	1.46x	9.1%
`correct_color`	0.93s	0.154s	6.04x	37.8%
copy to output	0.25s	0.047s	5.32x	33.3%
overall	1.97s	0.49s	4.02x	25.1%

It is to note that this table has runtime values without vectorization, with vectorization the overall efficiency climbs to about 50%.

From this table we can conclude that our optimization is not perfectly scalable, but we have a better utilization of resources than what we started with. This is especially noticeable ate higher iteration counts.

We also tried other amounts of threads (less due to processor limits) and these were the results:

CycleId	Iterative	OpenMP (8 T)	Speedup	Efficiency
uchar calc.	0.22s	0.040s	5.5x	68.8%
`gray_img`	0.29s	0.061s	4.75x	59.4%
histogram	0.07s	0.080s	0.875x	10.9%
`correct_color`	0.93s	0.162s	5.74x	71.8%
copy to output	0.25s	0.035s	7.14x	89.3%
overall	1.97s	0.55s	3.58x	44.8%

4 CONCLUSIONS:

In conclusion we improved the runtime of the program significantly by applying the parallelization and vectorization techniques. This resulted in a increase of resource efficiency of about 50% overall, and made plausible the processing of large or high quality images with more iterations.

We can also conclude that running this program on 8 CPU Threads rather than running it on 16, increases the efficiency and makes the program overall more scalable, meaning that although runtime might be larger we also utilize less resources.

5 INDIVIDUAL CONTRIBUTIONS:

Diogo Pinto:

(1) Implementation of OpenMP version (45%);
(2) Ran tests and made scripts for testing (5%);

David Mira:

(1) Implementation of CUDA version (45%);
(2) Report writing and data analysis (5%);

6 WHAT COULD BE BETTER:

We now realize that our initial analysis was very simple, and that we could've analyzed how changes in the flow of execution would affect the efficiency of the program, or how the access to memory could be further optimized. These would be some of the things we would've liked to have done, but didn't get to.

REFERENCES

[1] Google. 2024. GoogleTest. https://github.com/google/googletest
[2] JetBrains IDEs. 2024. Learn CLion. https://www.jetbrains.com/clion/learn
[3] NVIDIA. 2024. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/
[4] OpenMP. 2024. OpenMP. https://www.openmp.org/
[5] Hervé Paulino. [n. d.]. GPU Programming. ([n. d.]). Divided into lectures 11 and 12.