Iterative Histogram Equalization

André Singh 67107
Gonçalo Gingeira 60355
a.singh@campus.fct.unl.pt
g.gingeira@campus.fct.unl.pt

1 INTRODUCTION

This report documents the parallelization and optimization of an iterative histogram equalization implementation for the Concurrency and Parallelism course. Histogram equalization is a fundamental image processing technique that enhances image contrast by redistributing pixel intensities for better visual clarity. In the report we will describe our optimizations and compare the results with the sequential implementation.

2 ITERATIVE HISTOGRAM EQUALIZATION

2.1 Image Processing

This project focuses on the iterative histogram equalization of digital images. An image is represented as a two-dimensional array of pixels, where each pixel holds an intensity value. Histogram equalization aims to redistribute these intensity values to achieve a more uniform distribution, enhancing the image's contrast. Iterative approaches refine this process over multiple steps. The codebase implements a sequential version of iterative histogram equalization. During each iteration, the image's histogram is calculated, a cumulative distribution function (CDF) is derived, and intensity values are remapped. This process is repeated until a convergence criterion is met or a certain number of iterations is completed.

2.2 Histogram Subdivision

Iterative histogram equalization operates sequentially, with each iteration refining the results of the previous one. This dependence prevents parallelization across iterations. To achieve performance gains, we focus on optimizing the internal steps within each iteration. The histogram equalization function exhibits the following key sections, offering promising targets for parallelization:

(1) **Image Conversion**: Transform the floating-point input image into an unsigned character array and a grayscale array.
(2) **Histogram Calculation**: Compute the distribution of pixel intensities within the grayscale image.
(3) **CDF Calculation**: Derive the cumulative distribution function (CDF) from the computed histogram.
(4) **Image Correction**: Apply the calculated CDF to remap pixel intensities in the uchar image, enhancing contrast.

2.3 Initial Measure

To start to optimize our program, we decided to use the perf profiler in order to measure the sequential code and compare the weight of each function compared to the histogram calculation.

Image conversion:	23.85%
Compute Histogram:	3.38 %
Compute CDF:	0.04%
Image Correction Output:	72.59%

2.4 Performance Metrics used in our project

In the evaluation of parallel computing systems, several critical metrics are utilized to assess performance improvements and resource utilization:

- **Execution Time on a Single Processor (T_1):**
  - *Description:* The time taken to complete a task using only a single processor, serving as a baseline for comparison.
- **Execution Time on a "p" Processor System (T_p):**
  - *Description:* The execution time when the same task is run on a system equipped with "p" processors.
- **Speedup $\frac{T_1}{T_p}$:**
  - *Description:* The improvement in execution time with p processors compared with 1 processor
- **Efficiency $\frac{S(p)}{p}$:**
  - *Description:* Measures how well the computational capacity is utilized when adding more processors

3 OPTIMIZATIONS

3.1 Test Environment

For the first part of the project we ran the code in a machine with the following hardware : AMD Ryzen 5000 Series, 16 GB RAM and for the second part of the project we ran the code in a machine with the hardware: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz, 8GB RAM and NVIDIA GeForce MX130.

Initial measures: Initially to determine our measures and enhacements we essentially used perf profiler and chrono functions, latter on we used other tools like plotting of graphs with python pandas and making use of aggregation of data on a .csv before hand.

Code Section	Time to run 100 iterations on output3000x3000.ppm file
Image Conversion	29.45s
Histogram Calculation	5.2s
CDF Calculation	0.00014s
Image Correction Output	62.62s
Total Time to Iterate	92.22s

Table 1: Iterations Time Table

3.2 Optimization 1

In this optimization we parallelized the loop in our function that converts float to char with a pragma for. By using this implementation each pixel can be processed independently. The execution time in the version with one thread was 6.56s.

Performance	8 Threads	12 Threads	16 Threads
Execution Time	0.83s	0.58s	1.54s
Speedup	7.89	11.36	4.25
Efficiency	0.98	0.95	0.27

3.3 Optimization 2

Also in image conversion, we parallelized the nested loops in the function that converts the image to grey scale with a collapse(2). Since we have 2 nested loops one right after the other this function will distribute iterations of both loops across threads. Using something like only parallel for would work but wouldn't be as efficient as this solution. The execution time of the version with 1 thread was 4.30s.

Performance	8 Threads	12 Threads	16 Threads
Execution Time	0.73s	0.59s	1.026s
Speedup	6.02	7.31	4.21
Efficiency	0.75	0.61	0.26

3.4 Optimization 3

For this optimization we decided to use a reduction for the array because it ensures that the histogram array is correctly accumulated in parallel without race conditions. We previously tried to use an simple parallel for in order to parallelize the cycle and use atomic in histogram but this reduction seemed to work better because the atomic barely did any change in execution time and some times it even make it worse. The version with 1 thread takes 1.83s.

Performance	8 Threads	12 Threads	16 Threads
Execution Time	0.35s	0.19s	0.43s
Speedup	5.17	9.31	4.28
Efficiency	0.64	0.78	0.27

3.5 Optimization 4

Now in the function we used a single in order to not use multiple threads to solve this loop, which if used would make the execution time bigger We started to parallelizable the first loop. The total time to run this with 1 thread was 0.00018s.

Performance	8 Threads	12 Threads	16 Threads
Execution Time	0.00025s	0.0064s	0.29s
Speedup	0.495	0.028	0.0006
Efficiency	0.088	0.023	3.04584e-05

3.6 Optimization 5

For this optimization we merged 2 functions together. Since one loop calculated the values for the image array and the following loop used those values, we would traverse the same loop twice which affected the complexity of our program. With this optimization we merged them and applied a parallel optimization which improved a lot the image correction section of the code. The sequential part of the program took 26.13s to run.

Performance	8 Threads	12 Threads	16 Threads
Execution Time	6.53s	2.76s	5.40s
Speedup	3.99	9.47	4.83
Efficiency	0.50	0.79	0.30

4 OPTIMIZATION CONCLUSION

The results obtained from the various optimizations implemented during the project highlight the significant improvements in performance through parallelization techniques. Here's a breakdown of the findings:

- The parallelization of the image conversion and histogram calculations using OpenMP significantly reduced execution times across different thread configurations. Notably, a higher number of threads generally resulted in better performance, demonstrating effective use of system resources.
- The application of a reduction strategy in the histogram calculation improved data handling efficiency, effectively managing potential race conditions and enhancing the computation speed.
- Efficiency metrics varied across optimizations, with some configurations achieving near-optimal efficiencies. This variance highlights the importance of fine-tuning thread allocations and parallelization strategies based on the specific computational demands and data dependencies of each task.

Overall, these optimizations underscore the crucial role of parallel processing techniques in enhancing computational performance.

5 PERFORMANCE COMPARISON

Utilizing the script "tests.sh" to collect data and Python's Pandas library for data analysis (testing.py), we generated the graph depicted bellow to compare the performance of our program across different threading configurations. The graph distinctly illustrates the execution times as we varied the number of threads used in the program.

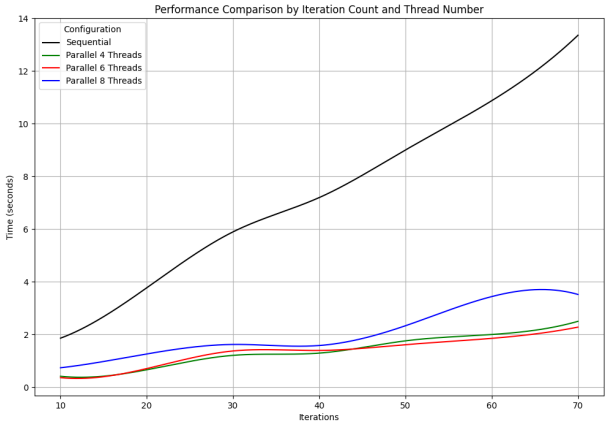

Figure 1: Comparative Measurements.

In our comparative analysis of the program's performance under different thread configurations, we observed distinct trends in execution time as a function of the number of threads utilized. Specifically, the program's performance improves significantly as the number of threads increases from 4 to 6. This enhancement can be attributed to more efficient parallel processing and better utilization of CPU resources.

However, when the thread count exceeds 6, we noted a plateau and subsequent deterioration in performance. This degradation beyond 6 threads could be due to several factors such as increased overhead from thread management, resource contention among threads, or limitations in the hardware since my colleague managed to get better results with 12 threads. The results suggest that our application achieves optimal performance with 6 threads, beyond which the overhead associated with managing additional threads outweighs the benefits of parallel execution.

The sequential version of the program, used as a baseline, shows a steady increase in execution time with the number of iterations, highlighting the benefits of parallel processing in reducing execution time for fixed iteration counts. Compared to sequential execution, parallel processing with up to 6 threads maintains significantly lower execution times across all tested iteration counts, demonstrating effective concurrency in our application.

5.1 Performance Comparison with GPU

In order to make this comparison we made a script were the results would obtain a .csv file with the "Image Size", "Number of threads", "GPU Time" and "CPU time" the tests we're all made for 500 iterations, as mentioned previously we used NVIDIA GeForce MX130. The images were also generated by a script but the caption of the size of each image is on the graph.

ATENTION: As performance improves exponentially and not linearly, we had to change the visualization of y axis to logarithmic scale in order to facilitate visualization and interpretation of the graph so it may seam like the differences between the GPU and CPU are big but if we didn't use logarithmic scale we would only see a line in the bottom of the GPU times and a line on top of the CPU times. The same size of images are represented with the same color changing only the "X" (CPU) and "Circle" (GPU) form in the lines. The amount of threads per block used for GPU comparison was the optimal value of 256 threads which we obtained from the CUDA occupancy calculator (explained on section 5.3)

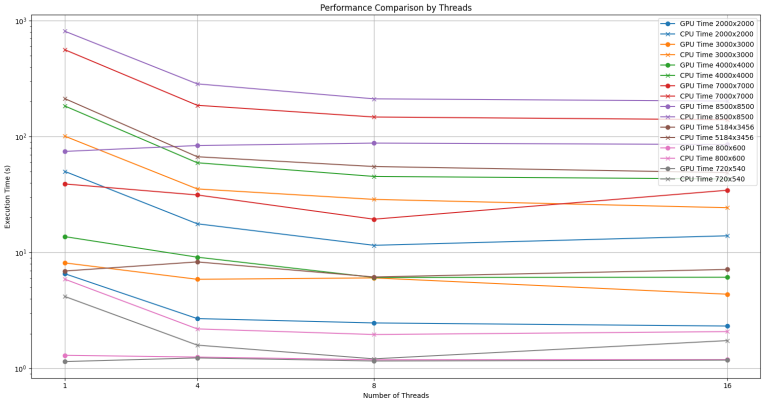

Figure 2: Comparative GPU-CPU Measurements.

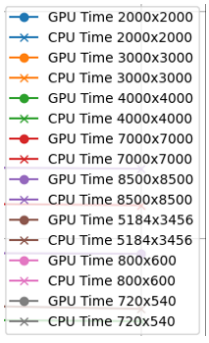

Figure 3: Label zoomed in

5.2 Pinpoints on the performance comparison:

- The graph shows how performance of the CPU implementation scales with the number of threads, i.e, when the number of threads increases, the execution time decreases, which was our initial goal.
- The GPU shows a better performance improvement the larger the image is, as you can see by the graph, it may seem that the intervals in time are the same the bigger the image is but since our time has a logarithmic scale the differences in performance for each image are actually quite big, for example, while the best performance for CPU time in 2000x2000 was 11,5 seconds and for GPU time was 2,5 seconds which means a difference of 9 seconds in 8500x8500 the best performance in CPU time was 202 seconds and for GPU time 83 seconds which makes it a difference of 119 this indicates that the GPU architecture is better suited for high-throughput and data-intensive tasks.

5.3 CUDA Occupancy calculator

The CUDA Occupancy Calculator is a tool provided by NVIDIA that allows developers to understand and optimize the occupancy of their CUDA kernels. Occupancy is a measure of how many threads the GPU is executing simultaneously relative to the number of threads it is capable of executing simultaneously. Higher occupancy generally indicates better utilization of the GPU cores, although it doesn't always correlate directly to higher performance.

the calculator is an excel provided by CUDA tools, where we open and provide with several metric which will help us understand the level of ocupancy such as:

- Compute capability, shared memory size and CUDA version (these we're obtained trough some research such as the page NVIDIA **page** of the GPU used
- Threads per Block
- Registers per Thread for each kernel call - We obtained this information trough the CUDA compiler output "Xptxas="v"
- Shared memory per block
- Block Size

After filling all the metrics with the necessary information for each kernel call we were presented with this graph:

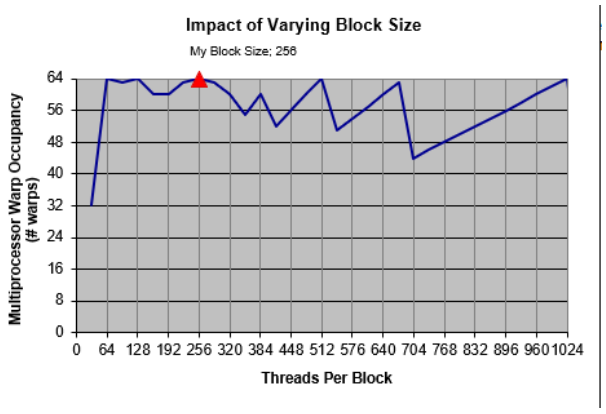

Figure 4: Impact of varying block size

The graph shows a significant decline in occupancy from smaller to mid-range block sizes, with more fluctuations observed as block size increases. Key observations are:

- Initial sharp decline in occupancy as the block size increases from small values up to around 128-256 threads.

- A plateau and slight fluctuations in occupancy for block sizes beyond 256, indicating variability in how well these sizes utilize GPU resources.
- A marked point at 256 threads per block shows optimal occupancy without excessive resource usage.

Based on the analysis, Configurations around 128, 256, and 512 threads per block are the most efficient setup.

We are aware that optimizing CUDA block sizes is not just about achieving high occupancy but also about balancing resource utilization to maximize overall application performance.

5.4 Relation between GPU and CPU Time Comparisons and CPU Metrics

When referring to our earlier formula, if we make an unconventional comparison by using CPU time as T_1 and GPU time as T_p, this isn't a direct application of the speedup formula but rather an illustrative comparison to demonstrate relative performance:

$$\text{Comparative Speedup} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}$$

This calculation is not about traditional speedup but rather comparing how much faster a GPU can perform the same task compared to a CPU, highlighting the differences in processing architectures and efficiencies. This type of analysis is particularly useful in demonstrating the advantages of GPUs for parallelizable tasks.

These metrics, speedup and efficiency, play a crucial role in optimizing and designing systems that are both powerful and cost-effective, enabling developers and researchers to make informed decisions about hardware and software strategies for handling complex computational tasks.

A bit NVIDIA on GeForce MX130: The type of GPU used is an important aspect on the performance of our GPU implementation, the MX130 is based on the Maxwell architecture with a maximum os 1024 threads per block, since the warp size is 32 (which is common among all NIVIDA GPUs) we used multiples of 32 to test which pair of (BlockWidth, BlockHeight) had the best performance in our program we made 10 tests to different images and 100 iterations and registred the best results:

Image Size	Block Size (8, 8)	Block Size (8, 16)	Block Size (16, 16)	Block Size (32, 32)
4000x4000	3.55s	3.59s	3.51s	3.58s
8500x8500	27.85s	27.92s	27.77s	28.05s
7000x7000	13.2833s	13.410s	13.367s	17.4813s

Table with calculations for speedup and efficiency for GPU-CPU:

Threads	CPU Time (s)	CPU Speedup
1	811.86	1.00
4	285.05	2.85
8	211.74	3.83
16	202.56	4.01

Table 2: CPU Performance Metrics

Threads	CPU Efficiency	Comparative Speedup ($\frac{T_{\text{CPU}}}{T_{\text{GPU}}}$)
1	1.00	10.89
4	0.71	3.40
8	0.48	2.41
16	0.25	2.37

Table 3: Performance Metrics for GPU and CPU

Comparative Speedup: The "Comparative Speedup" metric, which measures the performance ratio of CPU time to GPU time ($\frac{T_{\text{CPU}}}{T_{\text{GPU}}}$), offers significant insights into the differential scaling capabilities of CPUs and GPUs. Initially, the GPU outperforms the CPU significantly when only one thread is involved, demonstrating a comparative speedup of approximately 10.89 times. This superior performance by the GPU can be attributed to its architecture, which is inherently designed to handle multiple operations simultaneously and efficiently, making it ideally suited for parallel processing tasks.

The CUB functions like HistogramEven are designed to abstract away many of the lower-level details of CUDA programming, including the optimization of grid and block sizes. CUB internally determines the optimal launch configuration based on the operation it needs to perform and the hardware capabilities of the GPU. This means that we dont need to manually specify blocks per grid and threads per block for CUB functions, however we tested

Initially we made a kernel for each function in the code, after an opinion from the Professor we came to the conclusion that aggregating for's would minimize the number of kernel and thus calls to the global memory, for example the function "combinedGPU" calls the kernel "combinedKernel" which is a junction of both conversion to grayscale and convert float to Uchar

6 CONCLUSION

This report has detailed the methodology and results of our efforts to enhance the performance of an iterative histogram equalization process through various optimization techniques in a parallel computing environment. The primary objective was to decrease execution times and improve resource utilization, thereby demonstrating the practical benefits of parallel processing in image enhancement tasks.

6.1 Key Findings

Our findings confirm that parallelization significantly reduces the computation time required for histogram equalization by distributing workloads across multiple processors. The optimizations implemented at different stages of the histogram equalization process not only improved execution times but also highlighted the potential for scaling these approaches to larger data sets and more complex image processing tasks.

- **Optimal Thread Utilization:** We identified that the optimal number of threads for our application was 12, beyond which the performance gains diminished. This threshold was crucial for maintaining efficiency without incurring excessive overhead from thread management and synchronization.
- **GPU vs. CPU Performance:** The GPU consistently outperformed the CPU in terms of execution time, particularly with larger images. This underlines the GPU's superior capabilities in handling data-intensive and parallelizable tasks, making it an ideal choice for high-throughput image processing applications.
- **Scalability and Efficiency:** The speedup and efficiency calculations revealed that while speedup increased with the number of threads, efficiency tended to decrease, indicating a trade-off between fast execution and resource utilization.

6.2 Implications

The implications of this study are twofold. Firstly, it underscores the importance of careful thread and resource management in parallel computing to achieve optimal performance. Secondly, it suggests that for applications similar to histogram equalization, GPUs offer significant advantages in terms of speed and efficiency over CPUs, particularly as the size and complexity of the task increases.

7 CONTRIBUTIONS

The distribution of tasks between the team members, is outlined as follows:

Setup do projeto e análise inicial

- André: 70%
- Gonçalo: 30%

Implementação da paralelização e identificação de candidatos de paralelização: (Stage 1)

- André: 70%
- Gonçalo: 30%

GPU acceleration (Stage 2)

- André: 100%
- Gonçalo: 0%

Testes e ajustes finais

- André: 70%
- Gonçalo: 30%

Documentação e report

- André: 50%
- Gonçalo: 50%

8 BIBLIOGRAPHY

REFERENCES

[1] Professor Hervé Paulino, Lecture slides on parallel computing.
[2] Muatik GitHub Repository, https://github.com/muatik/openmp-examples - Example usage of OpenMP.
[3] NVIDIA CUB Library, https://github.com/NVIDIA/cub - All primitives for CUB.
[4] Danny Ruijters GitHub Repository, https://github.com/DannyRuijters/CubicInterpolationCUDA - Usage of CUDA.
[5] Microsoft, https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170 - For pragma and OpenMP directives.
[6] Learn C++, https://www.learncpp.com/ - C++ basics.
[7] NVIDIA, https://developer.nvidia.com/blog/even-easier-introduction-cuda/ - Stage 2.
[8] NVIDIA, https://developer.nvidia.com/how-to-cuda-c-cpp - Stage 2.
[9] YouTube, https://www.youtube.com/watch?v=f1YNyQqbiAM - CUDA - Stage 2.
[10] YouTube, https://www.youtube.com/watch?v=FQ1k_YpyG_A - Some videos on the playlist.
[11] NVIDIA CCCL, https://nvidia.github.io/cccl/cub/ - Device wide primitives.
[12] NVIDIA, https://www.nvidia.com/pt-br/drivers/nvidia-geforce-mx130/ - Information page on GeForce MX130.

9 ANNEX

9.1 Performance GPU-CPU Data

Image Size	Threads	GPU Time (s)	CPU Time (s)
2000x2000	1	6.576982259750366	50.18725085258484
2000x2000	4	2.6963818073272705	17.718873500823975
2000x2000	8	2.4732518196105957	11.550484895706177
2000x2000	16	2.3256194591522217	13.952128887176514
3000x3000	1	8.149657011032104	101.03217053413391
3000x3000	4	5.882972478866577	35.396806955337524
3000x3000	8	6.047399997711182	28.753965139389038
3000x3000	16	4.37553858757019	24.37337565422058
4000x4000	1	13.731564283370972	184.03881788253784
4000x4000	4	9.13134503364563	59.50980615615845
4000x4000	8	6.100465297698975	45.35458779335022
4000x4000	16	6.126968622207642	43.15042734146118
7000x7000	1	39.07607698440552	562.1133477687836
7000x7000	4	31.397982835769653	186.34470415115356
7000x7000	8	19.405315160751343	147.62391686439514
7000x7000	16	34.487576723098755	139.9650444984436
8500x8500	1	74.56124711036682	811.8601236343384
8500x8500	4	83.80800676345825	285.04857206344604
8500x8500	8	87.96692609786987	211.7392566204071
8500x8500	16	85.51656937599182	202.5649905204773
5184x3456	1	6.932292461395264	212.98239517211914
5184x3456	4	8.31010627746582	67.06103992462158
5184x3456	8	6.158467769622803	55.22846961021423
5184x3456	16	7.153079271316528	48.907490491867065
800x600	1	1.300508975982666	5.900558948516846
800x600	4	1.2555246353149414	2.1974079608917236
800x600	8	1.1908724308013916	1.9651708602905273
800x600	16	1.1948697566986084	2.0825307369232178
720x540	1	1.1485645771026611	4.187431812286377
720x540	4	1.234682559967041	1.591792345046997
720x540	8	1.1616244316101074	1.21250581741333
720x540	16	1.1818716526031494	1.740997314453125

9.2 Performance CPU

$test_type$	iterations	threads	$time_seconds$
sequential	10	1	1.584928096
parallel	10	4	.259900400
sequential	10	1	73.848695423
parallel	10	4	10.867415425
sequential	10	1	1.572244394
parallel	10	4	.275118999
sequential	10	1	1.652568893
parallel	10	6	.248060699
sequential	10	1	75.237912678
parallel	10	6	13.312906981
sequential	10	1	1.491281901
parallel	10	6	.204426300
sequential	10	1	1.622071502
parallel	10	8	.184466800
sequential	10	1	78.480435225
parallel	10	8	13.952746506
sequential	10	1	1.636550860
parallel	10	8	.351220891