Histogram Equalization with CUDA

Rozhina Ahmadi

Jan Antón Villanueva

TGA - Graphic Cards and Accelerators Project 2024-25Q2

Agustín Fernández

Daniel Jiménez

GitHub Repository: https://github.com/rozhinaahmadii/tga_hist_equalization

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Contents

1. Introduction

This project explores the use of CUDA to accelerate histogram equalization to improve contrast. We implement and evaluate several CUDA versions, comparing them against a CPU baseline across various image sizes and GPU configurations (1, 2, and 4 GPUs).

Input Images

To test scalability and performance, we used a variety of image resolutions and complexities:

1. **IMG00**: 1366 × 876 pixels
2. **IMG01**: 5184 × 3456 pixels
3. **IMG02:** 5184 × 3456 pixels
4. **IMG03**: 3072 x 1728 pixels
5. **IMG04**: 3072 x 1728 pixels

CPU Baseline

The baseline implementation was the provided CPU version, which executes the full histogram equalization pipeline entirely on the host. To ensure a fair comparison with GPU versions, we measured only the execution time of the eq_CPU() function, explicitly excluding image loading and saving operations.

Each implementation was assigned a unique code ID for reference; the CPU version was labeled as ID 1. For speedup calculations, we selected two images of different resolutions and used their CPU execution times as the reference for evaluating all single-GPU versions.

ID	CPU/GPU	IMG	Equalization Time (ms)	Version Name
1	CPU	IMG00	35.022	Hist_CPU.cu
1	CPU	IMG01	207.44	Hist_CPU.cu

Table 1 CPU Execution Times

2. Single-GPU CUDA Versions

We developed 5 different CUDA versions with varying memory and concurrency strategies. All of them follow a row-wise kernel approach.

Implemented Versions					
ID	CPU/GPU	Kernel Strategy	Memory Type	Streams	Version Name
2	1 GPU	Rowwise	Global	No	eq_rowwise_global.cu
3	1 GPU	Rowwise	Shared (Standard)	No	eq_rowwise_shared_standard.cu
4	1 GPU	Rowwise	Shared (Pinned)	No	eq_rowwise_shared_pinned.cu
5	1 GPU	Rowwise	Shared (Pinned)	Yes	eq_rowwise_shared_pinned_streams.cu
6	1 GPU	Rowwise	Shared (Standard)	Yes	eq_rowwise_shared_standard_streams.cu

Table 2 Single-GPU Cuda Versions

2.1 Single-GPU Results IMG01 (all versions)

We tested all the previous versions with the IMG01 as input to find the version with the best performance.

All CUDA-based implementations were benchmarked using cudaEventElapsedTime() to measure individual kernel durations. For each version, we measured and summed the following GPU operations:

1. RGB → YCbCr conversion with initial histogram
2. Blur on the Y channel
3. Histogram using shared memory
4. Equalization and reconstruction back to RGB

The total kernel time was computed as the sum of these measured stages, excluding any image I/O overhead.

PerformanceIMG01 (ms)						
ID	RGB → YCbCr	Blur Y channel	Histogram time	Equalize time	Total kernel time	Total runtime
2	5.313	0.356	5.077	1.01	11.576	300.634
3	5.3	0.359	0.242	1.011	6.912	295.688
4	5.272	0.351	0.243	1.006	6.871	15.161
5	2.004		0.435	1.042	3.481	291.425
6	1.843	0.349	0.233	1.008	3.433	289.721

Table 3 Performance Breakdown (in milliseconds)

To evaluate performance improvement, we define speedup as:

$$Speedup = \frac{T\text{CPU}}{T\ GPU}$$

Where:

1. T_{CPU}: Execution time of eq_CPU() (excluding I/O)
2. T_{GPU}: Total kernel time from the CUDA version (also excluding I/O)

ID	CPU	Total kernel time	Speedup
2	207.444	11.576	17.92018
3	207.444	6.912	30.01215
4	207.444	6.871	30.19124
5	207.444	3.481	59.59322
6	207.444	3.433	60.42645

Table 4 Speedup Calculations

The best performance was obtained with version 6, which utilizes shared memory, the standard memory model, and CUDA streams, achieving a speedup of 60.43x.

However, to support multi-GPU execution, which is incompatible with the use of streams, we needed a version without stream-based concurrency. Therefore, we compared:

1. Version 3: Shared memory with standard memory allocation, no streams
2. Version 4: Shared memory with pinned memory allocation, no streams

Version 3 had a total kernel time of 6.912 ms, while Version 4 slightly outperformed it with 6.871 ms. Given this minor performance gain and its compatibility with multi-GPU setups, version 4 was selected for continued development of multiple-GPU versions.

2.2 Single-GPU Results IMG00 (version 4)

As with IMG01, we tested the best-performing version (ID 4) on a smaller image (IMG00). The total kernel time was 0.800 ms, compared to 35.022 ms on the CPU, resulting in a speedup of 43.78x.

PerformanceIMG00 (ms)						
ID	RGB → YCbCr	Blur Y channel	Histogram time	Equalize time	Total kernel time	Total runtime
4	0.633	0.044	0.033	0.09	**0.8**	2.437

Table 5 Performance for Image IMG00 and ID4

ID	CPU	Total kernel time	Speedup
4	35.022	0.8	**43.7775**

Table 6 Speedup for Image IMG00 and ID4

This confirms that the chosen implementation performs consistently across different image sizes, although the speedup is slightly different.

3. Multi-GPU Implementation

To test how the algorithm scales across multiple GPUs, we extended the best-performing single-GPU version (eq_rowwise_shared_pinned.cu) to support execution on 2 and 4 GPUs. This version was chosen because it gave good performance and didn't use CUDA streams, which can be more complex to manage across devices.

In the multi-GPU setup, we used cudaSetDevice() to assign each GPU a portion of the image. Each device handled its own memory allocation and kernel execution. The image was divided horizontally into equal parts, and each GPU processed one segment independently using the same row-wise kernel.

CPU/GPU	Kernel Strategy	Memory Type	Streams
2 GPUs	Rowwise	Shared (Pinned)	No
4 GPUs	Rowwise	Shared (Pinned)	No

Table 7 Multi-GPU Cuda Versions

3.1 Results with Speedups

We evaluated the performance across different image sizes using 1, 2, and 4 GPUs. We measured kernel execution time using cudaEventElapsedTime(), excluding any I/O. Speedup was calculated as:

$$Speedup = \frac{1 - GPU\ Kernel\ Time}{N - GPU\ Kernel\ Time}$$

Image	GPUs	Kernel Time (ms)	Speedup
IMG00	1	0.8	
	2	2.489	0.32×
	4	2.267	0.35×
IMG01	1	6.871	
	4	16.094	0.43×
IMG02	1	7.182	
	4	15.478	0.46×
IMG02	1	1.907	
	2	5.775	0.33×
IMG04	1	1.888	
	2	5.695	0.33×
	4	5.785	0.33×

Table 8 Multi-GPU Results and Speedups

3.2. Results Analysis

In all our tests, using more than one GPU actually slowed down the program. The speedup was always below 1, meaning it was less efficient than using just one GPU. This effect was more noticeable with smaller or medium-sized images.

We expected that 2 or 4 GPUs would reduce the execution time, but instead, the overhead of splitting the data and managing multiple devices outweighed the benefits. This is a common issue when the workload per GPU is not large enough. In our case, the following factors likely contributed:

1. Overhead in splitting the image: Sending separate chunks to each GPU added time. For relatively small images, this extra cost was greater than the gains from parallel execution.
2. No overlapping of operations: Since we didn't use CUDA streams or asynchronous copies, memory transfers and kernel executions were done sequentially.
3. Workload imbalance: We divided the image evenly by rows, assuming equal effort for each GPU. But this doesn't always result in a balanced workload, especially if the number of rows isn't divisible evenly or if GPUs have different performance.
4. Pinned memory across devices: Although pinned memory speeds up transfers, using it independently on multiple GPUs required separate allocations and transfers, which added overhead.

Multi-GPU applications often use additional techniques such as one CPU thread per GPU, streams for overlapping data transfers, and unified memory. These were beyond the scope of this project but could help improve scalability in more advanced implementations.

4. Conclusion

In this project, we implemented and analyzed several CUDA-based versions of the histogram equalization algorithm and compared their performance to a CPU baseline.

Our single-GPU implementations showed a significant speedup over the CPU version. The best-performing version used shared memory with CUDA streams and achieved up to 60x speedup on large images. Even for smaller images, the speedups remained high, confirming that GPU acceleration can greatly reduce processing time when memory and execution are optimized.

We also extended the implementation to 2 and 4 GPUs. However, instead of improving performance, these versions ran slower. This was mainly due to the overhead of splitting the image, managing separate memory allocations, and the lack of overlapping between transfers and kernel execution. These results show that multi-GPU setups do not always scale well, especially when the workload is not large or complex enough to benefit from parallel distribution.

Overall, this project helped us understand how GPU performance depends on both the algorithm and the system setup. CUDA provided a major improvement over CPU processing, and we learned that sometimes, a well-optimized single GPU is more effective than trying to scale across multiple devices.