UNIVERSITÀ DEGLI STUDI FIRENZE

Histogram Equalization

Parallel Computing Second Course Project

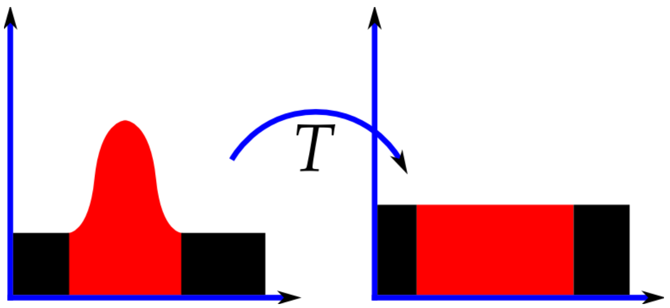
Implementation and Performance Analysis of Sequential and Parallel Version in CUDA

Giovanni Stefanini

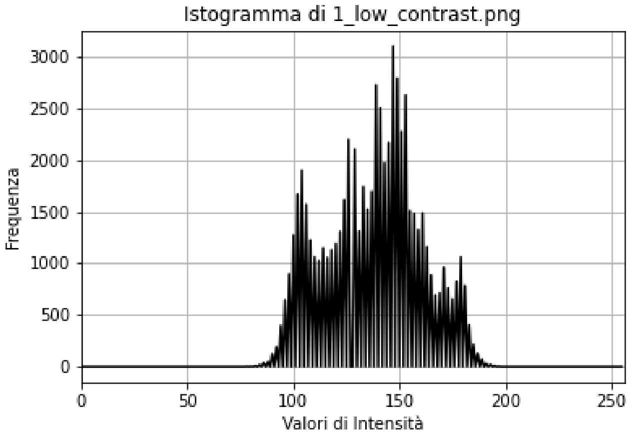
Firenze 2024-2025

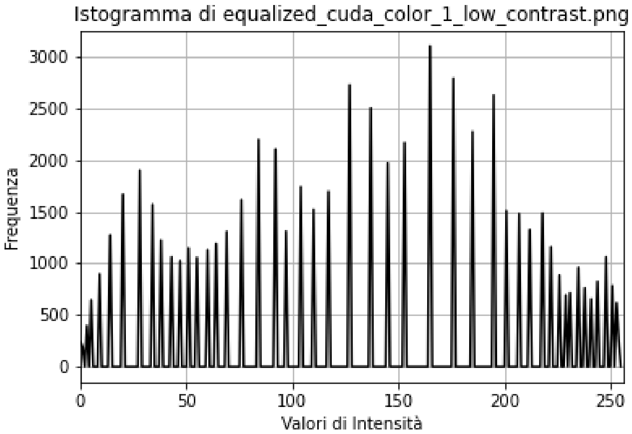
Introduction

Histogram Equalization

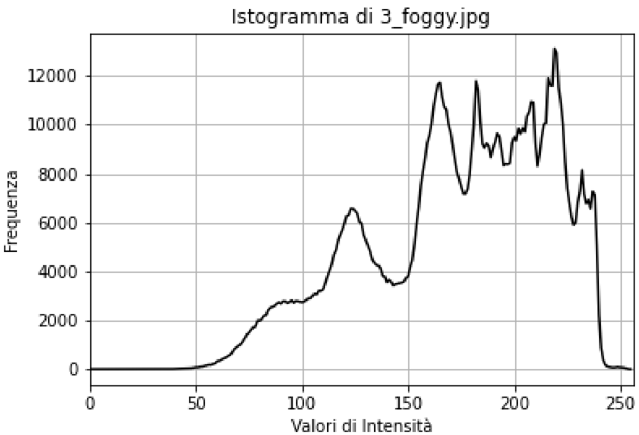


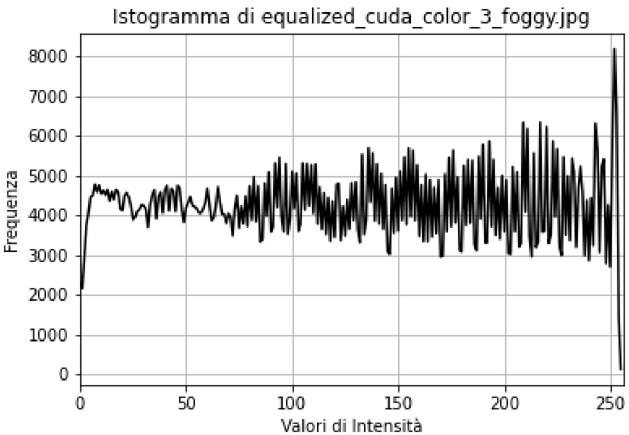
- **Histogram Equalization** is a technique for **enhancing image contrast**.
- It **redistributes the intensity levels** of an image's histogram to achieve a more uniform distribution.
- It uses the **Cumulative Distribution Function (CDF)** to remap intensity values.
- Applicable to both **grayscale** and **color images**.
- Useful in the preprocessing of **low-contrast** images to improve visual quality.





Grayscale image equalization result





Color image equalization result

Implementation

Sequential version

- Sequential version implemented in **C++** with **OpenCV**.

- **Main steps** of the sequential algorithm:
  - **Input**: Image (grayscale or color).
  - Check if the image is grayscale; otherwise, convert it to grayscale.
    1. Compute **histogram**.
    2. Compute **CDF**.
    3. **Normalize CDF** to obtain new pixel values.
    4. Apply **transformation** (pixel remapping).
  - **Output**: Equalized grayscale image.

Observations

Sequential version

- **Slow for large images** due to **nested loops over rows and columns** for ***histogram computation*** (step 1) and ***transformation application*** (step 4).

- ***CDF computation*** (step 3) is well-suited for parallel **reduction**, as it is obtained by progressively summing the histogram values.

- Therefore, a **parallel CUDA implementation** can improve efficiency by using **three kernels** to replace the corresponding steps of the sequential version.

Implementation

Parallel version CUDA

- Parallel version implemented in **CUDA** to exploit the GPU.

- Parallelized algorithm using **3 kernels**:
    1. ***computeHistogram***: computes **histogram** using tiling.
    2. ***computeCDF***: computes **CDF** using reduction.
    3. ***applyTransformation***: applies **transformation** using tiling.

- The parallel code includes **memory management** (CPU-GPU), **pinned memory allocation** to speed up data transfers, and **execution time measurement**.

- CDF normalization is performed on **CPU**.

NVIDIA®

Implementation

Memory Management CUDA

- Some objects are allocated using ***cudaMalloc*** in the GPU global memory:
  - A buffer for **input image**, with ***cudaMemcpy*** used to transfer the image from host to device.
  - A buffer for **output image**, with ***cudaMemcpy*** used to transfer the image from device to host.
  - An integer array of size 256 for **histogram**.
  - An integer array of size 256 for **CDF**.
  - A buffer of size 256 for **lookup table**.

- Some objects are allocated using ***cudaHostAlloc*** in the CPU memory:
  - An integer array of size 256 for **CDF**, with ***cudaMemcpyAsync*** used to copy values from GPU to CPU.
  - A buffer of size 256 for **lookup table**, with ***cudaMemcpyAsync*** used to copy values from GPU to CPU.

Implementation

First Kernel CUDA: computeHistogram

- Its purpose is to **compute histogram** in parallel.

- **Main steps**:
  1. **Initialize** local histogram to zero.
  2. **Load pixels** from global memory into shared memory (*__shared__ uchar tile[16][16]*).
  3. **Update local histogram** using atomic operations.
  4. **Merge with global histogram** using '*atomicAdd()*'.

- **Strategy**:
  - **Shared memory**: each thread block has shared memory for a local histogram, reducing access latency. The drawback is that local copies must be merged.
  - **Tiling**: Reduces global memory traffic by enabling **coalesced access**, improving efficiency.

Implementation

Second Kernel CUDA: computeCDF

- Its purpose is to **compute Cumulative Distribution Function (CDF)** in parallel.

- **Main steps**:
  1. Load **histogram into shared memory**.
  2. Use **parallel reduction** to progressively sum the values. Each value is updated by adding the previous one with an offset that doubles at each iteration.
  3. Write the result to global memory.

- **Strategy**:
  - ➢ **Shared memory**: for fast access.
  - ➢ **Reduce parallelo**: computing CDF requires **O(log N)** operations instead of **O(N)**.

Implementation

Third Kernel CUDA: applyTransformation

- Its purpose is to **apply pixel transformation** using precomputed lookup table.

- **Main steps**:

  After CDF normalization by the CPU, the lookup table is obtained.

  1. Load **data into shared memory** using **tiling** for fast access.
  2. **Each thread reads a pixel** and replaces it with the corresponding value from the lookup table.

- **Strategy**:
  - **Shared memory**: Each thread block has shared memory for a local histogram, reducing access latency. The drawback is that local copies must be merged.
  - **Tiling**: Enables **coalesced memory access**, maximizing GPU efficiency.

Experiments

Technical Specifications

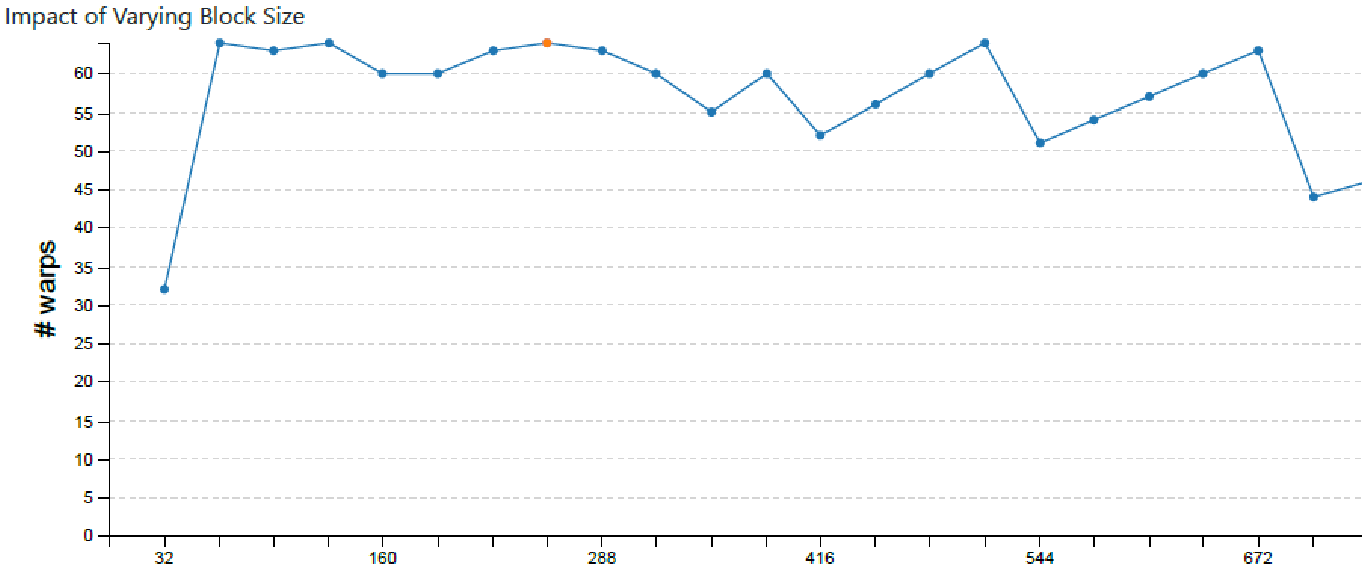
- **Hardware:**
  - **CPU**: **Intel Core i7-7700HQ** (2.8 GHz)
  - **GPU**: **NVIDIA GeForce GTX 1050**
  - RAM: 16 GB
  - Compute Capability: 6.1

- **Software:**
  - SO: Microsoft Windows 10 Home
  - **CUDA Compiler**: **nvcc 12.8.61**
  - **Host Compiler**: **MSVC 1942**
  - IDE: CLion 2024.2.2

Experiments

Technical Choices

- Analyzing ***occupancy*** and based on experiments, the following dimensions were chosen:
  - ***blockSize:* 16x16**, to have **256 threads per block**.
  - ***girdSize:*** depends on the image dimensions:
    *[(**width** + blockSize.x-1)/blockSize.x], [(**height** + blockSize.y-1)/blockSize.y)]*



Experiments

Images used

- Test **images** were selected to analyze **speedup** and **effectiveness** based on different **resolutions** and **lighting conditions**.



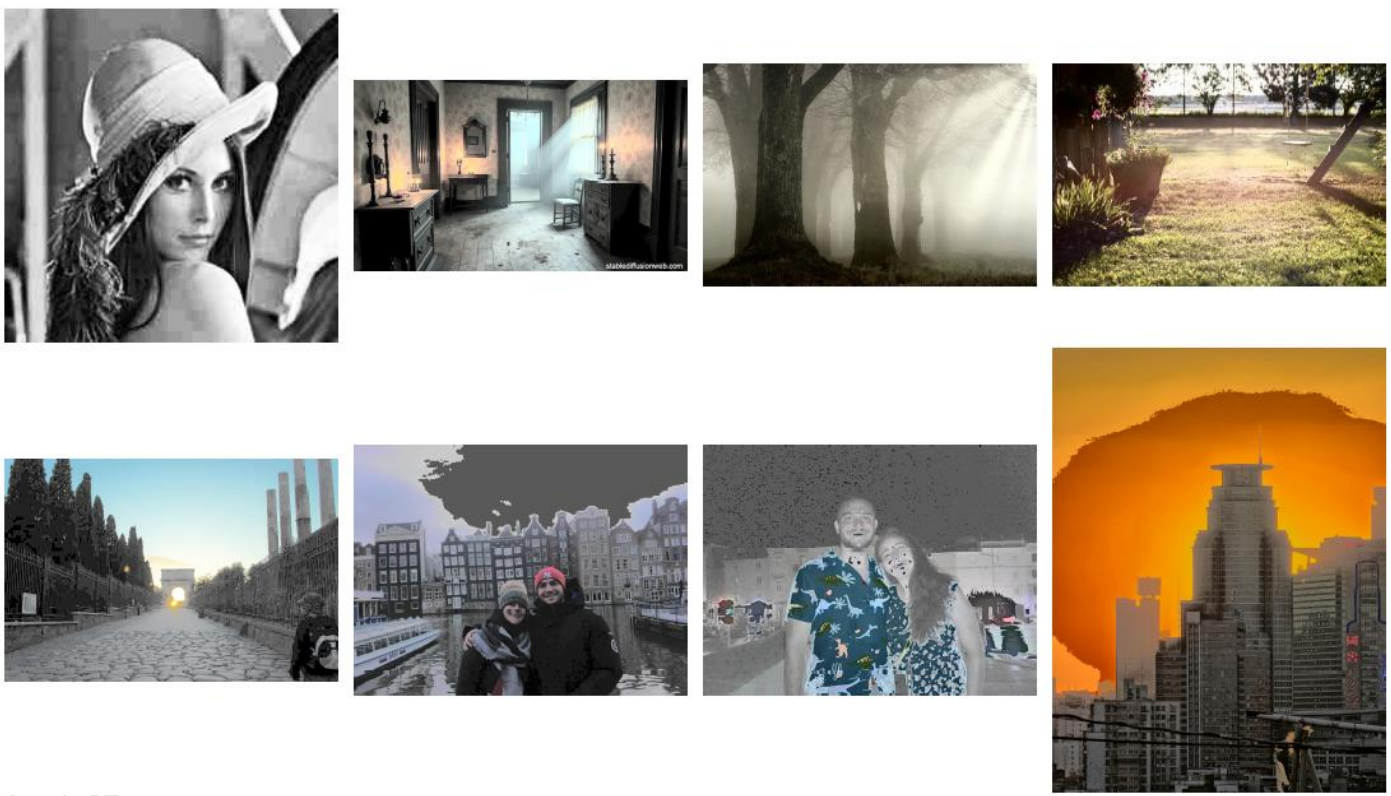





Results

Sequential images

- Visual result obtained with **sequential** Histogram Equalization, **excellent** for **low resolution** images, **terrible** for **high resolution** images.



Results

CUDA Images

- Visual result obtained with **CUDA parallel** Histogram Equalization, **great** for **low resolution** images as well as **high resolution** images.








Results

Execution Times

- For the parallel version, the **kernel execution time** (T_{kernel}) and the **total parallel execution time** (T_{cuda}) are reported, including data transfer times between CPU and GPU.

Immagine	$T_{sequenziale}$ (ms)	T_{kernel} (ms)	T_{cuda} (ms)
lc 256x256	2.1704	0.233248	89.3771
di 1344x768	34.5987	0.580288	78.435
f 1280x813	36.0089	0.575712	91.6379
o 608x406	8.3431	0.450464	89.9615
u 1024x683	22.8673	0.63056	89.523
o 3680x2760	359.179	3.46342	599.038
u 3680x2760	335.684	3.52854	636.197
hr 3072x4096	421.064	4.29763	564.916

Results

Speedup Analysis

- Notice how **Theoretical Speedup** (S_{kernel}) and **Actual Speedup** (S_{cuda}) vary.

Immagine	S_{kernel}	S_{cuda}
lc 256x256	9.31	0.024
di 1344x768	59.6	0.44
f 1280x813	62.5	0.39
o 608x406	18.5	0.093
u 1024x683	36.3	0.26
o 3680x2760	103.7	0.6
u 3680x2760	95.13	0.53
hr 3072x4096	97.99	0.75

- **Theoretical Speedup** (S_{kernel}): up to 100x for high resolution images.
- **Actual Speedup** (S_{cuda}): lower due to data transfer.

Conclusions

Analysis of results

- The **CUDA kernel offers** **high efficiency**, with speedups between 9x and 100x compared to the CPU, demonstrating the **superiority and scalability of parallel computing**, especially for high-resolution images.

- **Images processed with CUDA** look **visually better**.

- However, the **total execution time increases** significantly **due to** the **data transfer overhead between CPU and GPU**, which particularly affects small images.

- For larger images, such as 3680x2760 or 3072x4096, the overhead impact is reduced, allowing for an overall performance improvement. So, the **GPU** really becomes **advantageous for large images**.