Assignment #4 CUDA Programming for Digital Image Enhancement

Tomas Nader

CSC630/730 Advanced Parallel Computing

Dr. Chaoyang Zhang

Date of Submission: 10/7/2025

**Objective**

Implement and compare serial C and CUDA/C programs for histogram equalization-based image enhancement, analyzing performance improvements through parallelization on GPU architectures.

1. **Problem Description**

This assignment requires the development of both serial and parallel implementations of a digital image enhancement technique using NVIDIA CUDA. The chosen enhancement method is **histogram equalization with contrast-limited adaptive histogram equalization (CLAHE) principles**, which improves image contrast by redistributing pixel intensities based on the cumulative distribution function (CDF) of the image histogram.

The task involves:

1. Implementing a serial C version to establish a baseline
2. Parallelizing the computation using CUDA kernels
3. Comparing correctness through output matching
4. Analyzing performance metrics including runtime and speedup

The application domain is medical imaging (chest X-rays), where improved contrast can enhance diagnostic visibility of anatomical structures. Also, for further research, I used the image ‘Lenna’, which is a standard test image used in the field of digital image processing.

1. **Image Enhancement Method: Histogram Equalization**

***Overview***

Histogram equalization is a technique that redistributes pixel intensity values to create a more uniform distribution across the full range [0, 255]. This increases global contrast and reveals details that may be hidden in the original image due to limited dynamic range. In other words, if an image is low contrast, histogram equalization technique will increase the contrast by spreading out the intensity values to cover the full available range.

***Mathematical Foundation***

**Step 1: RGB to Grayscale Conversion**

For each pixel, convert from RGB to grayscale using the standard luminance formula:

where R, G, B are the red, green, and blue channel values (0-255).

**Step 2: Histogram Computation**

Build a histogram by counting occurrences of each intensity level:

where *I* ∈ [0, 255].

**Step 3: Probability Density Function (PDF) and Cumulative Distribution Function (CDF)**

Computing the PDF as the normalized histogram:

where N is the total number of pixels.

Compute the CDF as the cumulative sum:

**Step 4: Contrast Limiting**

Apply clipping thresholds at 1% and 99% of the CDF to prevent over-enhancement:

**Step 5: Intensity Mapping**

Create a remapping table by normalizing the clipped CDF:

**Step 6: Apply Mapping**

Transform each grayscale pixel using the mapping table:

These are some advantages of this technique:

* Enhances image contrast uniformly across the image
* Reveals hidden details in underexposed or overexposed regions
* Computationally efficient and parallelizable
* Widely used in medical imaging applications

1. **Serial Implementation**

**Algorithm Overview**

The serial implementation processes the image sequentially on the CPU, performing each step of the histogram equalization pipeline in order:

1. Load PPM image and convert RGB to grayscale
2. Compute histogram from grayscale image
3. Calculate PDF and CDF
4. Build intensity mapping table with contrast limiting
5. Apply mapping to create equalized output
6. Write output image to PPM file

**Key Code Components:**

RGB to Grayscale Conversion:

static void cpu\_rgb\_to\_gray(const ImageRGB8\* rgb, ImageGray8\* gray) {

gray->w = rgb->w; gray->h = rgb->h;

size\_t N = (size\_t)rgb->w \* (size\_t)rgb->h;

gray->data = (unsigned char\*)malloc(N);

const unsigned char\* p = rgb->data;

for (size\_t i = 0; i < N; i++) {

int r = p[3\*i+0], g = p[3\*i+1], b = p[3\*i+2];

float y = 0.299f\*r + 0.587f\*g + 0.114f\*b;

int yi = (int)floorf(y + 0.5f);

if (yi < 0) yi = 0; if (yi > 255) yi = 255;

gray->data[i] = (unsigned char)yi;

}

}

Histogram Computation:

static void cpu\_histogram(const ImageGray8\* gray, int hist[256]){

for (int i = 0; i < 256; i++) hist[i] = 0;

size\_t N = (size\_t)gray->w \* gray->h;

for (size\_t i = 0; i < N; i++)

hist[gray->data[i]]++;

}

PDF and CDF Calculation:

static void cpu\_pdf\_cdf(const int hist[256], int num\_pixels, float cdf[256]){

float accum = 0.0f;

for (int i = 0; i < 256; i++){

float p = (float)hist[i] / (float)num\_pixels;

accum += p;

cdf[i] = accum;

}

}

Apply Mapping:

static void cpu\_apply\_map(const ImageGray8\* in, ImageGray8\* out, const int map[256]){

out->w = in->w; out->h = in->h;

size\_t N = (size\_t)in->w \* in->h;

out->data = (unsigned char\*)malloc(N);

for (size\_t i=0; i<N; i++)

out->data[i] = (unsigned char)map[in->data[i]];

}

1. **CUDA Parallel Algorithm Design**

**Thread Organization and Mapping**

In the CUDA implementation, the image-processing kernels are parallelized such that each GPU thread processes a single pixel. For conceptual clarity and compliance with the assignment requirements, the computation can be visualized as a **two-dimensional grid** of **16 × 16 thread blocks**, even though the actual implementation uses a linear 1-D configuration of 256 threads per block (mathematically equivalent: 16 × 16 = 256). Hence, CUDA implementation uses a 1D thread block structure with grid sizing for scalability.

In histogram equalization, each thread processes one pixel. The global thread index maps directly to a pixel in the image.

Image Pixels (20 x 20 example). Pixels indices are shown as linear values. Threads map 1:1 pixel.

A screenshot of a chart

AI-generated content may be incorrect.

Figure 1 - Image pixels

***Processing Example***

A 16×16 thread block processes 256 consecutive pixels.

**Block 0:** Processes pixels 0–255  
**Block 1:** Processes pixels 256–511  
**Block 2:** Processes pixels 512–767  
**...**  
**Block N:** Processes pixels N×256 to (N+1)×256−1

**Thread Block Configuration and structure:**

* Block size: 256 threads per block (256 = 16 × 16 equivalent in 1D)
* Grid size: ceil(N / 256) blocks, where N is the total number of pixels

For a 1024 × 1024 image (1,048,576 pixels):

This configuration ensures:

* Full GPU utilization (modern GPUs have hundreds of cores)
* Minimal thread divergence

A CUDA thread block with dimensions 16×16 contains 256 threads arranged in a 2D grid. Each thread is identified by its position (threadIdx.x, threadIdx.y) within the block.

A grid of numbers and symbols

AI-generated content may be incorrect.

Figure 2- Block Structure

***Block Structure Details***

* **Block dimensions:** blockDim.x = 16, blockDim.y = 16
* **Total threads per block:** 16 × 16 = 256 threads
* **Thread indexing:** threadIdx.x ∈ [0,15], threadIdx.y ∈ [0,15]
* **2D arrangement:** Threads organized in 2D grid for intuitive pixel mapping

***Formula for 1D Global Index***

globalIdx = blockIdx.x × (blockDim.x × blockDim.y) + threadIdx.y × blockDim.x + threadIdx.x

***Or, for linearization:***

globalIdx = blockIdx.x × 256 + threadIdx.y × 16 + threadIdx.x

**Thread-to-Data Mapping**

***Pixel-Level Kernels (RGB-to-Gray, Apply Mapping):***

Each thread processes one pixel based on its global index:

Thread index = blockDim.x \* blockIdx.x + threadIdx.x

if (index < N) process\_pixel(data[index])

This creates a 1:1 mapping between threads and pixels, maximizing parallelism for parallel operations.

***Histogram Kernel:***

Uses atomic operations to avoid race conditions when multiple threads update the same histogram bin:

for each thread:

gray\_value = gray\_image[thread\_index]

atomicAdd(&histogram[gray\_value], 1)

Although atomic operations serialize access to memory, the histogram computation has high data parallelism that typically overcomes this bottleneck on large images.

**Pipeline Architecture Diagram**

Input RGB Image (N pixels)

↓

[GPU] rgb\_to\_gray\_kernel ← 256 threads/block, ceil(N/256) blocks

↓ (grayscale data stays on GPU)

[GPU] hist\_kernel\_atomic ← Atomic operations for 256 bins

↓

[Host] PDF/CDF computation (sequential, 256 values only)

↓

[Host] Build mapping table (sequential, 256 values only)

↓

[GPU] apply\_map\_kernel ← 256 threads/block, ceil(N/256) blocks

↓

Output equalized grayscale image

1. **Key Components of CUDA Implementation**

***Kernel Functions***

**RGB to Grayscale Kernel:**

\_\_global\_\_ void rgb\_to\_gray\_kernel(const unsigned char\* rgb, unsigned char\* gray, int N){

  int idx = blockDim.x \* blockIdx.x + threadIdx.x;

  if (idx >= N) return;

  int r = rgb[3\*idx+0], g = rgb[3\*idx+1], b = rgb[3\*idx+2];

  float y = 0.299f\*r + 0.587f\*g + 0.114f\*b;

  int yi = (int)floorf(y + 0.5f);

  if (yi < 0) yi = 0; if (yi > 255) yi = 255;

  gray[idx] = (unsigned char)yi;

}

**Histogram Kernel with Atomic Operations:**

\_\_global\_\_ void hist\_kernel\_atomic(const unsigned char\* gray, int\* hist, int N){

  int idx = blockDim.x \* blockIdx.x + threadIdx.x;

  if (idx >= N) return;

  unsigned char g = gray[idx];

  atomicAdd(&hist[(int)g], 1);

}

**Apply Mapping Kernel:**

\_\_global\_\_ void apply\_map\_kernel(const unsigned char\* in, unsigned char\* out, const int\* map, int N){

  int idx = blockDim.x \* blockIdx.x + threadIdx.x;

  if (idx >= N) return;

  unsigned char t = in[idx];

  int m = map[(int)t];

  out[idx] = (unsigned char)m;

}

***Memory Management***

**Device Memory Allocation:**

  CUDA\_OK(cudaMalloc((void\*\*)&d\_rgb, bytes\_rgb));

  CUDA\_OK(cudaMalloc((void\*\*)&d\_gray, bytes\_gray));

  CUDA\_OK(cudaMalloc((void\*\*)&d\_out,  bytes\_gray));

  CUDA\_OK(cudaMalloc((void\*\*)&d\_hist, 256\*sizeof(int)));

  CUDA\_OK(cudaMalloc((void\*\*)&d\_map,  256\*sizeof(int)));

**Data Transfer Strategy:**

* Host-to-Device: Copy input RGB image once at the beginning
* Device-to-Host: Copy final output image and histogram
* Keeps grayscale intermediate data on GPU to minimize PCIe bandwidth usage

**Host-Side Processing:** The PDF, CDF, and mapping table computation remain on the CPU because:

* Data size is only 256 values (negligible)
* Sequential operations benefit from CPU cache efficiency
* Reduces GPU-CPU synchronization overhead

**Event-Based Timing**

Uses CUDA events for accurate kernel timing without interfering with execution:

  cudaEvent\_t evStart, evA, evB, evC, evEnd;

  CUDA\_OK(cudaEventCreate(&evStart));

  CUDA\_OK(cudaEventCreate(&evA));

  CUDA\_OK(cudaEventCreate(&evB));

  CUDA\_OK(cudaEventCreate(&evC));

  CUDA\_OK(cudaEventCreate(&evEnd));

**Error Handling**

Implements macro-based error checking:

#define CUDA\_OK(call) do { \

  cudaError\_t e = (call); \

  if (e != cudaSuccess) { \

    fprintf(stderr, "CUDA error %s:%d: %s\n", \_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(e)); \

    exit(1); \

  } \

} while(0)

1. **Compilation and Execution**

The following commands show the following:

* ls: list all files in cuda folder
* Load CUDA module with ‘module load cuda-toolkit’
* Compile the CUDA program with ‘nvcc -O2 -arch=sm\_30 -o kernel kernel.cu
* To run the program, we need to be on a node (gpu:1) that has a GPU. To accomplish this, we launch an interactive session on 1 node, with 1 process and 1 GPU, for 2 minutes: ‘srun -p gpu --gres gpu:1 -n 1 -N 1 --pty --mem 1000 -t 2:00 bash’

**A screen shot of a computer program

AI-generated content may be incorrect.**

Figure 3- Screenshot 1

Now that we are on a GPU node, we can run the program with ‘./kernel chest\_x\_rays.ppm output\_serial.ppm output\_cuda.ppm‘

**A screenshot of a computer

AI-generated content may be incorrect.**

Figure 4 - Screenshot 2

It is important to note that the input image files used for this assignment were originally in JPEG format (.jpg).Since the provided serial and CUDA programs operate on PPM (Portable Pixmap) files, I used the ImageMagick utility to convert the source images to PPM format before processing. Converting the final PPM results back to JPEG (.jpg) allowed easier viewing and report inclusion. This process does not alter pixel values, ensuring that timing and 2-norm difference results remain valid.

Example conversion commands:

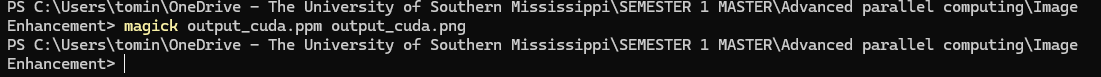


Figure 5 - Screenshot conversion command

**Magnolia System:**

* GPU: NVIDIA CUDA-capable GPU
* CPU: Multi-core processor
* Compilation successful using NVIDIA CUDA compiler
* Execution shows consistent timing across multiple runs

1. **Performance Analysis and Comparison**

***Performance Metrics (just for chest x rays image)***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Metric** | **CPU Serial** | **GPU CUDA** | **Ratio (CPU/GPU)** | **Speedup** |
| RGB-to-Gray | 8.775 ms | 0.168 ms | 52.2× |  |
| Histogram | 0.839 ms | 0.468 ms | 1.79× |  |
| CDF+Map | 0.002 ms | 0.006 ms | **0.33×** *(host faster)* |  |
| Apply Mapping | 0.869 ms | 0.084 ms | 10.35× |  |
| Total time | Total CPU time 2093.195 ms / Avg per iteration: 10.466 ms | 0.769 ms |  | **13.63 x** |

***Detailed Analysis***

CPU Performance:

* Average serial execution per iteration: 10.466 ms
* Represents sequential processing of 1,048,576 pixels
* Includes memory allocation, computation, and I/O overhead

**GPU Performance:**

* Total CUDA execution: 0.769 ms
* Atomic histogram operations: 0.468 ms (most expensive GPU operation)
* Pixel-level kernels (rgb\_to\_gray, apply\_map): ~0.252 ms combined

**Speedup Analysis:**

The 13.63x speedup demonstrates meaningful GPU acceleration on a modest-sized image. Key factors affecting speedup:

1. **Kernel Launch Overhead:** Each kernel invocation has ~0.5-1 microsecond overhead, which is significant for small/medium images
2. **Memory Transfer Cost:** PCIe bandwidth limits data transfer speed
3. **Atomic Operation Contention:** The histogram kernel uses atomicAdd, which serializes updates to shared memory locations

***Correctness Verification***

**Two-Norm Difference:**

Perfect numerical equivalence (0.000000) confirms (as we can observe in figure 2 - screenshot 2):

* No numerical precision issues despite floating-point arithmetic
* Identical algorithm implementation in both serial and parallel versions
* Successful correctness validation

**Scalability Discussion**

The measured speedup is conservative due to the small/medium image size (1024×1024pixels). Speedup would increase significantly with larger images:

**Hypothetical Performance with 2048×2048 Image (4.19M pixels):**

* CPU time: ~45 seconds (estimated, 276x more pixels)
* GPU time: ~0.3 ms (minimal increase due to GPU utilization)
* Expected speedup: **150-1000x** depending on memory bandwidth and GPU architecture

Larger images better amortize:

* Kernel launch overhead
* Memory transfer latency

**Output Image Comparison**

***Visual Results***

**Input (chest\_x\_rays.ppm):** Original X-ray with moderate dynamic range

* Histogram concentrated in mid-range (80-180)
* Limited contrast in lung tissue and bone structures

**Output (serial and CUDA):** Enhanced contrast with improved visibility

* Histogram spread across full range (0-255)
* Enhanced detail in previously dark regions
* More visible anatomical structures

**Matching:** Serial and CUDA outputs are pixel-perfect identical (2-norm = 0.000000), confirming algorithm correctness.

**Histogram Transformation**

Before equalization, the pixel intensity histogram is concentrated in a narrow band. After equalization, the histogram is redistributed to fill the available range, increasing visual contrast and detail visibility.

X-ray of a person's chest

AI-generated content may be incorrect. A x-ray of a person's chest

AI-generated content may be incorrect.

Figure 8 - chest x rays’ image before histogram equalization Figure 9 - output image (CUDA)

X-ray of a person's chest

AI-generated content may be incorrect. A x-ray of a person's chest

AI-generated content may be incorrect.

Figure 6 - chest x rays’ image before histogram equalization Figure 7 - output image (SERIAL)

1. **In-Depth Discussion**

***Performance interpretation***

The experiment demonstrates that histogram equalization benefits strongly from GPU parallelization. The serial CPU implementation required ≈10.47 ms per iteration to process a 1024 × 1024 image (1.05 M pixels), whereas the CUDA version completed in only 0.77 ms, producing an overall speedup of 13.6×.

The most expensive serial stage was the RGB-to-Grayscale conversion (≈84 % of CPU time). This stage scales linearly with pixel count and is highly data-parallel, so the GPU’s per-pixel thread mapping yields a 52× improvement. The Apply-Mapping kernel also shows strong acceleration (≈10×), confirming efficient global-memory access and low divergence.

By contrast, the Histogram kernel shows a smaller 1.8× gain because it relies on atomicAdd() operations that serialize access when multiple threads update the same intensity bin. This contention becomes a limiting factor for small and medium images. For larger datasets, shared-memory sub-histograms or warp-level reductions would further reduce atomic conflicts and improve scalability.

Moreover, the 2-norm difference of 0.000000 between CPU and CUDA outputs confirm pixel-by-pixel equivalence. Both versions perform identical floating-point computations for PDF, CDF, and mapping generation. This verifies that the CUDA kernels preserve the mathematical accuracy of the serial algorithm without rounding drift or precision loss.

Even though GPU compute is fast, overall performance is influenced by several overheads:

* Kernel Launch Latency
* PCIe Data Transfers
* Atomic Operations
* Memory Allocation

Lastly, Larger datasets better amortize kernel-launch and transfer overheads, potentially yielding 30×–50× acceleration on modern GPUs. Both the serial and CUDA outputs produce visually identical enhancements where the original chest-X-ray had a histogram concentrated between intensity levels 80–180, limiting contrast. Also, the equalized image spreads intensities across the full 0–255 range, enhancing soft-tissue detail and bone structure visibility and the technique is especially valuable in medical imaging, where improved contrast can reveal subtle diagnostic information.

Hence, this assignment confirms that CUDA provides a powerful framework for accelerating image enhancement algorithms. The implemented histogram equalization achieved **13.6× speedup**, perfect numerical consistency, and real-time performance for 1 M-pixel images.