Histogram Equalization: Sequential and Parallel Version

Gianluca Giuliani
E-mail address
giuliani.gianluca1@edu.unifi.it

Abstract

Histogram Equalization is an image processing technique that enhance the contrast of images. The method redistributes the pixel intensity values in an image so that the resulting histogram is approximately uniform. Both sequential and parallel versions of histogram equalization are implemented, respectively in C++ and CUDA. The sequential in C++. The performance of both implementations are compared considering the speedup.

1. Device

The code is executed on an Intel I7 processor with 14 cores and 20 logic processors. The device code is executed on an NVidia GeForce 3070 Ti for a laptop with 6 GB of VRAM.

2. Algorithm

The K-means clustering algorithm is implemented using the following steps:

- Compute the Histogram:

  Calculate the histogram H that counts the occurrence of each intensity level i (ranging from 0 to L - 1, where L is the number of intensity levels, 256 for 8 bit images).

- Calculate the Cumulative Distribution Function (CDF):

  Compute the cumulative sum of the histogram values:

$$\text{CDF}(i) = \sum_{j=0}^{i} H(j)$$

  Normalize the CDF by dividing each value by the total number of pixels NN

- Map Intensities Using the Transformation Function:

  Define a transformation function T(i)T(i) based on the normalized CDF:

$$T(i) = \text{round}((L-1) \times \text{CDF}_{\text{norm}}(i))$$

  This maps the original intensity values i to new values T(i), redistributing them to span the entire range of possible intensities [0,L - 1].

  Replace each pixel intensity I(x,y) in the image with its corresponding transformed value T(I(x,y)).

Figure 1: Original grayscale image

Figure 2: Equalized image

Figure 3: Effect of equalization of histogram on Grayscale image

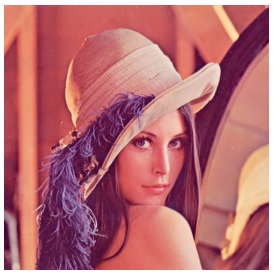

Figure 4: Original RGB image

Figure 5: Equalized image

Figure 6: Effect of equalization of histogram on RGB image

3. Sequential Code

3.1. Sequential.h

This code provides an image processing pipeline for histogram equalization using OpenCV. It handles both grayscale and color images, resizes them to predefined resolutions, and processes them to enhance contrast. The results and execution times are saved into a CSV file for analysis.

3.1.1 Histogram Computation

```
void computeHistogram(const Mat &channel,
int histogram[L]) {
    memset(histogram, 0,
        L * sizeof(int)); // Init histogram
    for (int i = 0; i < channel.rows; i++)
    {
        for (int j = 0; j <
            channel.cols; j++)
        {
            histogram[channel.
                at<uchar>(i,j)]++;
        }
    }
}
```

The function `computeHistogram()` calculates the frequency of each pixel intensity for a single-channel image. It initializes the histogram with zeros using `memset()` and iterates through the image to populate the histogram array.

3.1.2 CDF Computation

```
void computeCDF(const int histogram[L],
float cdf[L], int totalPixels) {
    cdf[0] = static_cast<float>
        (histogram[0]) / totalPixels;
    for (int i = 1; i < L; i++) {
        cdf[i] = cdf[i - 1] +

        static_cast<float>(histogram[i])/
                totalPixels;
    }
}
```

The function `computeCDF()` computes the cumulative distribution function (CDF) from the histogram. The CDF is normalized by dividing by the total number of pixels, ensuring that the values lie between 0 and 1.

3.1.3 Histogram Equalization:

```
void performHistogramEqualization(Mat &channel) {
    int histogram[L];
    float cdf[L];

    computeHistogram(channel, histogram);
    computeCDF(histogram, cdf,
        channel.rows * channel.cols);

    // Create a mapping for pixel
    // values based on the CDF
    uchar equalizedMap[L];
    for (int i = 0; i < L; i++) {
        equalizedMap[i] =
        static_cast<uchar>(255 * cdf[i]);
    }

    // Apply the mapping to the channel
    for (int i = 0; i < channel.rows; i++) {
        for (int j = 0; j < channel.cols; j++) {
            channel.at<uchar>(i, j) =
            equalizedMap[channel.at<uchar>(i, j)];
        }
    }
}
```

The function `performHistogramEqualization()` enhances the contrast of an image by remapping pixel intensities using the precomputed CDF. This mapping is applied to all pixels in the image, effectively improving its dynamic range.

Pipeline Structure:

- The `processImage()` function processes both grayscale and color images.
- Resolutions such as `1280x720` (HD), `1920x1080` (Full HD), and `3840x2160` (4K) are predefined.
- For grayscale images, histogram equalization is directly applied to the single channel.
- For color images, the image is first converted to the YCrCb color space. Histogram equalization is performed on the luminance (`Y`) channel, and the channels are merged back to recreate the enhanced color image.

Performance Measurement: Using the `chrono` library, the code measures the execution time of histogram equalization for different resolutions and writes the results to a CSV file. Each entry includes the resolution, execution time, pipeline type, and the number of channels (1 for grayscale and 3 for color).

Pipeline Details:

- **Grayscale Pipeline:**
  - Reads the input image in grayscale format using `IMREAD_GRAYSCALE`.

- Resizes the image to multiple resolutions and applies histogram equalization.

- **Color Pipeline:**
  - Reads the input image in color format using `IMREAD_COLOR`.
  - Converts the image to the YCrCb color space to perform histogram equalization only on the luminance (`Y`) channel.
  - Merges the processed luminance channel back with the original chrominance channels and converts the image back to BGR format.

4. Parallel Code

4.1. Kernel.cuh

4.1.1 Tiled Histogram Computation Kernel with Privatization

```
__global__ void computeHistogramTiled(const
    unsigned char* d_img, int* d_hist,
    int width, int height) {

    __shared__ int sharedHist[NUM_BINS];

    int tid = threadIdx.y * blockDim.x
        + threadIdx.x;
    int blockSize = blockDim.x *
        blockDim.y;

    // Initialize the shared histogram
    // in parallel
    for (int i = tid; i < NUM_BINS; i +=
        blockSize) {

        sharedHist[i] = 0;
    }
    __syncthreads();

    // Compute global pixel coordinates
    int x = blockIdx.x * blockDim.x +
        threadIdx.x;
    int y = blockIdx.y * blockDim.y +
        threadIdx.y;

    // Compute histogram in shared memory
    if (x < width && y < height) {
        int idx = y * width + x;
        atomicAdd(&sharedHist[d_img[idx]], 1);
    }
    __syncthreads();

    // Merge shared histogram to
    // global histogram
    for (int i = tid; i < NUM_BINS; i +=
        blockSize) {

        atomicAdd(&d_hist[i], sharedHist[i]);
    }
}
```

1. **Shared Memory as Private Copies:** Each block creates a private copy of the histogram in shared memory (`sharedHist[NUM_BINS]`). This minimizes contention for global memory, as threads within a block only interact with their own shared memory histogram.

2. **Initialization of Shared Memory:** The shared histogram bins are initialized to zero by the threads within the block. This ensures that each block starts with a clean, independent histogram. To ensure that all threads complete this initialization before moving forward (`__syncthreads()`) is used .

3. **Parallel Tile Processing:** Each thread processes a specific pixel in a tile (a subsection of the image) by using block and thread indices (`blockIdx` and `threadIdx`) to compute the global coordinates (`x`, `y`).

4. **Atomic Operations in Shared Memory:** The histogram is updated in shared memory using `atomicAdd()` to ensure that increments are performed safely without race conditions. Since shared memory has lower latency compared to global memory, this step is more efficient than directly modifying the global histogram.

5. **Shared-to-Global Histogram Reduction:** After processing the tile, the shared memory histogram is reduced into the global histogram (`d_hist`) by each thread in the block. Another `atomicAdd()` is used here to combine the contributions of all blocks into the final global histogram.

6. **Thread Synchronization:** Final synchronization (`__syncthreads()`) ensure consistent access to shared memory and prevent race conditions during both the initialization and reduction phases.

Privatization

- Reduces global memory access: Instead of updating the global histogram directly, each block processes its own private copy in shared memory, minimizing expensive global memory transactions.
- Exploits fast shared memory: Shared memory is significantly faster than global memory, making it ideal for localized, high-frequency operations such as histogram updates.

- Improves scalability: By privatizing the histogram to each block, the kernel can process large images efficiently while keeping contention over global memory low.

4.1.2 CDF computation

After computing the histogram using a CUDA kernel, the Thrust library is used to compute the cumulative distribution function (CDF) using an inclusive scan operation. The smallest CDF value (corresponding to the lowest non-zero intensity) is then used to normalize the CDF and generate a lookup table (LUT), which maps the original pixel intensities to their equalized values.

```
// Copy histogram into a thrust
// device vector.
thrust::device_vector<int>
    d_hist_vec(NUM_BINS);
CUDA_CHECK(cudaMemcpy(
thrust::raw_pointer_cast(
    d_hist_vec.data()),d_hist,NUM_BINS *
    sizeof(int),
    cudaMemcpyDeviceToDevice));

// Compute the CDF using an inclusive scan.
thrust::device_vector<int> d_cdf(NUM_BINS);
thrust::inclusive_scan(d_hist_vec.begin(),
d_hist_vec.end(), d_cdf.begin());

// Get the first value of the CDF
//(minimum non-zero value).
int cdf0 = d_cdf.front();

// Create a thrust device vector for the LUT.
thrust::device_vector<unsigned char>
    d_lut(NUM_BINS);

int threadsPerBlock = 256;
int blocksPerGrid = (NUM_BINS +
    threadsPerBlock - 1) / threadsPerBlock;
int* d_cdf_ptr = thrust::raw_pointer_cast(
    d_cdf.data());
unsigned char* d_lut_ptr =
    thrust::raw_pointer_cast(d_lut.data());

applyLutKernel<<<blocksPerGrid,
    threadsPerBlock>>>(d_cdf_ptr, d_lut_ptr,
    imgSize, cdf0, NUM_BINS);
CUDA_CHECK(cudaDeviceSynchronize());
```

4.1.3 LookUp Tables computations

```
// Device function to compute a single LUT
// value from a CDF value.
__device__ unsigned char computeLutValue(int cdf_val,
    int totalPixels, int cdf0) {
    float norm = (cdf_val - cdf0) /
        float(totalPixels - cdf0);
    return static_cast<unsigned char>(
        norm * 255.0f + 0.5f);
}

// Kernel that applies the computeLutValue
// function to each element of the CDF array.
__global__ void applyLutKernel(const int* d_cdf,
    unsigned char* d_lut, int totalPixels, int cdf0,
    int numBins) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < numBins) {
        d_lut[idx] = computeLutValue(d_cdf[idx],
            totalPixels, cdf0);
    }
}
```

ComputeLutValue computes the Lut values. First the CDF value is normalized and it scales the value in the range from [0,1] to [0,256]. ApplyLutKernel applies the computeLutValue() to each element of the CDF to obtain the Lut.

4.1.4 Histogram Equalization

Each pixel in the input image is read, and its intensity is replaced with the new intensity from the LUT. This produces the final equalized image.

$$\text{new\_value} = \frac{\text{CDF}[\text{value}] - \text{CDF}_{\text{min}}}{\text{CDF}_{\text{max}} - \text{CDF}_{\text{min}}} \times 255$$

```
__global__ void applyEqualization(
const unsigned char* d_img,
unsigned char* d_out,
const unsigned char* d_lut, int imgSize) {

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

if (idx < imgSize)
    d_out[idx] = d_lut[d_img[idx]];
}
```

5. Pipeline

To produce the results I define the processImageCuda() function.

```
int processImageCuda(std::string imgPath,
    std::string csvPath) {

// Warm-up dummyKernel<<<1, 1>>>();
CUDA_CHECK(cudaDeviceSynchronize());
```

```
// Load the original RGB image.
Mat originalRGB = imread("../imgs/
lena_rgb.png", IMREAD_COLOR);
...

// Convert RGB to grayscale once.
Mat originalGray;
cvtColor(originalRGB, originalGray,
    COLOR_BGR2GRAY);

// Define target resolutions
vector<Size> resolutions = { Size(1280, 720),
Size(1920, 1080), Size(3840, 2160) };

// Define a local lambda to process
// one resolution.
auto processResolution =
    [&](const Size &res) {

    ....
    Mat resizedGray, resizedRGB;
    resize(originalGray, resizedGray, res);
    resize(originalRGB, resizedRGB, res);
    runHistogramEqualizationPipelines(
    resizedGray, resizedRGB);
};

// Loop over each resolution.
for (const Size &res : resolutions) {
    processResolution(res);
}

return 0;

}
```

This function implements the pipeline that performs histogram equalization on an image at multiple resolutions.

1. **GPU Warm-Up:** A dummy kernel (`dummyKernel`) is launched to initialize the GPU. This step helps reduce the impact of any first-run overhead during subsequent processing.

2. **Image Loading:**
   The image is loaded using OpenCV's `imread`. The image is loaded in color mode (IMREAD_COLOR).

3. **Grayscale Conversion:**
   The loaded RGB image is converted to a grayscale image with `cvtColor`. This conversion is performed once and the result is stored in `originalGray`.

4. **Resolution Definition:**
   A vector of target resolutions is defined. The test resolutions are HD (1280x720), FullHD (1920x1080), and 4K (3840x2160).

5. **Processing via a Local Lambda:**
   A lambda function `processResolution` describes the processing for a single resolution. The lambda:

   - Resizes both the grayscale and RGB images to the target resolution.
   - Calls the `runHistogramEqualizationPipelines` function to execute the histogram equalization pipelines on the resized images both in grayscale case and rgb case.

5.1. Grayscale Pipeline

First the grayscale image is processed:

1. Allocate device memory for the input and output grayscale images.
2. Transfer the grayscale image from host to device memory.
3. Record the start event.
4. Execute the histogram equalization.
5. Record the stop event and synchronize.
6. Retrieve the elapsed time for performance evaluation.
7. Copy the processed image back to host memory.
8. Free device memory.

This approach ensures efficient execution of histogram equalization on the GPU while leveraging CUDA's parallel processing capabilities. The function that executes all the kernel described before id histogramEqualization().

```
float grayscaleTime = 0.0f;
float rgbTime = 0.0f;

cudaEvent_t start, stop;
CUDA_CHECK(cudaEventCreate(&start));
CUDA_CHECK(cudaEventCreate(&stop));

int graySize = grayImage.rows * grayImage.cols;
unsigned char *d_gray = nullptr,
    *d_grayEqualized = nullptr;

CUDA_CHECK(cudaMalloc(&d_gray, graySize *
    sizeof(unsigned char)));
CUDA_CHECK(cudaMalloc(&d_grayEqualized, graySize *
    sizeof(unsigned char)));
CUDA_CHECK(cudaMemcpy(d_gray, grayImage.data,
    graySize * sizeof(unsigned char),
     cudaMemcpyHostToDevice));

CUDA_CHECK(cudaEventRecord(start, 0));
```

```
    histogramEqualization(d_gray,
    d_grayEqualized, grayImage.cols,
    grayImage.rows);

CUDA_CHECK(cudaEventRecord(stop, 0));
CUDA_CHECK(cudaEventSynchronize(stop));
CUDA_CHECK(cudaEventElapsedTime(
    &grayscaleTime,
    start, stop));

Mat grayEqualized(grayImage.size(),
    grayImage.type());
CUDA_CHECK(cudaMemcpy(grayEqualized.data,
    d_grayEqualized, graySize *
    sizeof(unsigned char),
    cudaMemcpyDeviceToHost));

CUDA_CHECK(cudaFree(d_gray));
CUDA_CHECK(cudaFree(d_grayEqualized));
```

5.1.1 Histogram Equalization

Fist the memory for the histogram is allocated and initialized. Block and grid dimension are defined to do the execute the histogram computation kernel. Then we use thrust inclusive scan to compute the CDF and the LUT as already discussed. Finally the equalization is applied.

```
void histogramEqualization(unsigned char*
    d_in, unsigned char* d_out, int width,
    int height) {

    int imgSize = width * height;
    int *d_hist = nullptr;
    CUDA_CHECK(cudaMalloc(&d_hist,
        NUM_BINS * sizeof(int)));
    CUDA_CHECK(cudaMemset(d_hist, 0,
        NUM_BINS * sizeof(int)));

    // Launch kernel to compute histogram
    // using tiling.
    dim3 block(TILE_WIDTH, TILE_WIDTH);
    dim3 grid((width + TILE_WIDTH - 1) /
        TILE_WIDTH, (height + TILE_WIDTH
            - 1) / TILE_WIDTH);
    computeHistogramTiled<<<grid, block>>>(
        d_in, d_hist, width, height);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Copy histogram into a
    // thrust device vector.
    thrust::device_vector<int> d_hist_vec(
        NUM_BINS);
    CUDA_CHECK(cudaMemcpy(
        thrust::raw_pointer_cast(
        d_hist_vec.data()),d_hist,
        NUM_BINS * sizeof(int),
        cudaMemcpyDeviceToDevice));

    // Compute the CDF using an inclusive scan.
    thrust::device_vector<int> d_cdf(NUM_BINS);
    thrust::inclusive_scan(d_hist_vec.begin(),
        d_hist_vec.end(), d_cdf.begin());

    // Get the first value of the CDF
    // (minimum non-zero value).
    int cdf0 = d_cdf.front();

    // Create a thrust device vector
    // for the LUT.
    thrust::device_vector<unsigned char>
        d_lut(NUM_BINS);

    int threads = 256;
    int blocksPerGrid = (NUM_BINS + threads - 1)
        / threads;
    int* d_cdf_ptr = thrust::raw_pointer_cast(
        d_cdf.data());
    unsigned char* d_lut_ptr =
        thrust::raw_pointer_cast(d_lut.data());

    applyLutKernel<<<blocksPerGrid, threads>>>(
        d_cdf_ptr, d_lut_ptr, imgSize, cdf0, NUM_BINS);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Apply the equalization by mapping the original
    // image through the LUT.
    int blocks = (imgSize + threads - 1) / threads;
    applyEqualization<<<blocks, threads>>>(d_in,
        d_out, d_lut_ptr, imgSize);
    CUDA_CHECK(cudaDeviceSynchronize());

    CUDA_CHECK(cudaFree(d_hist));
}
}
```

5.1.2 RGB Pipeline

The RGB pipeline performs histogram equalization on a color image by processing only its luminance (Y) channel. The image is first converted from RGB to YCbCr, the Y channel is extracted and equalized using the same GPU-based histogram equalization function as in the grayscale pipeline, and finally the image is reconstructed back to RGB.

Packing the RGB Image: The RGB image is loaded into a host-side array of `uchar4`. Although the image is in RGB order, it is packed into a `uchar4` structure where the fourth component (alpha) is set to 255 (fully opaque). This allows efficient, vectorized memory transfers and coalesced access on the GPU.

```
int numPixels = rgbImage.rows * rgbImage.cols;
uchar4 *h_rgb = new uchar4[numPixels];
```

```
for (int i = 0; i < rgbImage.rows; i++) {
    for (int j = 0; j < rgbImage.cols; j++) {
        Vec3b pix = rgbImage.at<Vec3b>(i, j);
        h_rgb[i * rgbImage.cols + j] =
            make_uchar4(pix[0], pix[1], pix[2], 255);
    }
}
```

Allocating and Copying Data to the GPU: Device memory is allocated for the original RGB image (`d_rgb`), its YCbCr representation (`d_ycbcr`), and the final equalized RGB image (`d_rgbEqualized`). The host data is then copied to `d_rgb`.

```
uchar4 *d_rgb = nullptr, *d_ycbcr =
    nullptr, *d_rgbEqualized = nullptr;
CUDA_CHECK(cudaMalloc(&d_rgb, numPixels *
    sizeof(uchar4)));
CUDA_CHECK(cudaMalloc(&d_ycbcr, numPixels *
    sizeof(uchar4)));
CUDA_CHECK(cudaMalloc(&d_rgbEqualized,
    numPixels * sizeof(uchar4)));
CUDA_CHECK(cudaMemcpy(d_rgb, h_rgb,
    numPixels * sizeof(uchar4),
    cudaMemcpyHostToDevice));
```

Color Space Conversion (RGB to YCbCr): The `rgb2ycbcr` kernel is launched to convert each RGB pixel into its corresponding YCbCr value. This operation is parallelized over all pixels.

```
int threadsPerBlock = 256;
int blocksPerGrid = (numPixels +
    threadsPerBlock - 1) / threadsPerBlock;
CUDA_CHECK(cudaEventRecord(start, 0));
rgb2ycbcr<<<blocksPerGrid,
    threadsPerBlock>>>(d_rgb, d_ycbcr,
    numPixels);
CUDA_CHECK(cudaDeviceSynchronize());
```

Extracting the Y Channel: Since histogram equalization is applied only to the luminance channel, the `extractYChannel` kernel extracts the Y component from each YCbCr pixel and stores it in a separate buffer `d_Y`.

```
unsigned char *d_Y = nullptr,
    *d_YEqualized = nullptr;
CUDA_CHECK(cudaMalloc(&d_Y, numPixels *
    sizeof(unsigned char)));
CUDA_CHECK(cudaMalloc(&d_YEqualized,
    numPixels * sizeof(unsigned char)));
extractYChannel<<<blocksPerGrid,
    threadsPerBlock>>>(d_ycbcr, d_Y,
    numPixels);
CUDA_CHECK(cudaDeviceSynchronize());
```

Histogram Equalization on the Y Channel: The previously described `histogramEqualization` function is then applied to the extracted Y channel. This function computes the histogram, the cumulative distribution function (CDF), generates a lookup table (LUT), and remaps the pixel values in the Y channel.

```
histogramEqualization(d_Y, d_YEqualized,
    rgbImage.cols, rgbImage.rows);
```

Updating the Y Channel in the YCbCr Image: The `updateYChannel` kernel replaces the original Y channel in the YCbCr image with the equalized Y values.

```
updateYChannel<<<blocksPerGrid,
    threadsPerBlock>>>(d_ycbcr,
    d_YEqualized, numPixels);
CUDA_CHECK(cudaDeviceSynchronize());
```

Color Space Conversion (YCbCr to RGB): After updating the Y channel, the modified YCbCr image is converted back to RGB using the `ycbcr2rgb` kernel.

```
ycbcr2rgb<<<blocksPerGrid,
    threadsPerBlock>>>(d_ycbcr,
    d_rgbEqualized, numPixels);
CUDA_CHECK(cudaDeviceSynchronize());
CUDA_CHECK(cudaEventRecord(stop, 0));
CUDA_CHECK(cudaEventSynchronize(stop));
CUDA_CHECK(cudaEventElapsedTime(&rgbTime,
    start, stop));
```

Retrieving the Result and Cleanup: The equalized RGB image is copied back to the host and reconstructed as an OpenCV `Mat`. Finally, all allocated device memory is freed.

```
uchar4 *h_rgbEqualized = new uchar4[numPixels];
CUDA_CHECK(cudaMemcpy(h_rgbEqualized,
    d_rgbEqualized, numPixels *
    sizeof(uchar4), cudaMemcpyDeviceToHost));

Mat rgbEqualized(rgbImage.size(), CV_8UC3);
for (int i = 0; i < rgbImage.rows; i++) {
    for (int j = 0; j < rgbImage.cols; j++) {
        uchar4 pix = h_rgbEqualized[i *
            rgbImage.cols + j];
        rgbEqualized.at<Vec3b>(i, j) =
            Vec3b(pix.x, pix.y, pix.z);
    }
}

CUDA_CHECK(cudaFree(d_rgb));
CUDA_CHECK(cudaFree(d_ycbcr));
CUDA_CHECK(cudaFree(d_rgbEqualized));
CUDA_CHECK(cudaFree(d_Y));
```

```
CUDA_CHECK(cudaFree(d_YEqualized));
delete[] h_rgb;
delete[] h_rgbEqualized;
```

6. Data

Lena image was used at different resolution using Opencv.

7. Opencv usage

Opencv was used to obtain the desired resolution of the image and to read and show the images.

8. Experiments

The metric used to compare the performances of the sequential algorithm with the parallel CUDA version is the speedup, computed as

```
Speedup = ExecutionTime_seq /
    ExecutionTime_paral
```

8.1. Varying the resolution of the Images

The initial test aim to measure the speedup achieved with different resolutions of the same image. This is done both for the Grayscale and RGB version.

9. Results

A simple python script plots the speedup from the execution times in a csv file.

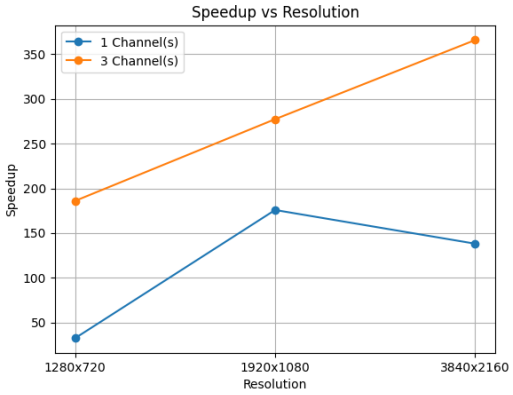

Figure 7: Experiment using tile of 32 and block 16

9.1. Profiling

The profiling of the code is done using NSight.

9.2. Conclusions

The reports describes how to implement Histogram equalization in c++ for the sequential version and c++ Cuda for the parallel version showing the speedup achieved.