Technical university „Gheorghe Asachi” Iasi
Faculty of Automatic Control and Computer Engineering
Distributed systems and web technologies

Histogram equalization

Students:

Dobreanu Mircea-Constantin

Chelaru Ștefan

Problem statement

Our assignment is to equalize the histogram of an input grayscale image, which has the effect of improving the contrast of the image. So the black regions of the original image become even more black and the white regions become even whiter.

CPU implementation

The basic algorithm is already implemented in an efficient way by the OpenCV library, and the basic flow of the algorithm is:

1. Compute histogram of the input image.
2. Compute the cumulative distribution function from the input image.
3. For each pixel of the original image assign a new intensity value based on its value in the original image and the cumulative distribution function.

GPU implementation

The same steps apply as above but we have to account for the parallel nature of computing on GPUs.

1. Compute histogram of the input image by counting the number of pixels of a specific intensity. This is done with the help of the atomic add operation provided by CUDA. Probably the biggest factor in execution time.
2. Compute the cumulative distribution function (prefix sums/inclusive scan) based on the Hillis-Steele algorithm. Basically there are log2(n) stages in which the values are propagated to cells that are at an offset distance where offset is a power of 2. The implementation is not necessarily efficient but since there are only 256 possible gray levels, the potential for performance improvements here is not that great.
3. Based on the pixel's value in the original image and the cumulative distribution, compute the new value in the improved image. This operation is inherently parallel.

Used images:

Image 1

512x384

Image 2

910x682

Image 3

1000x667

Image 4

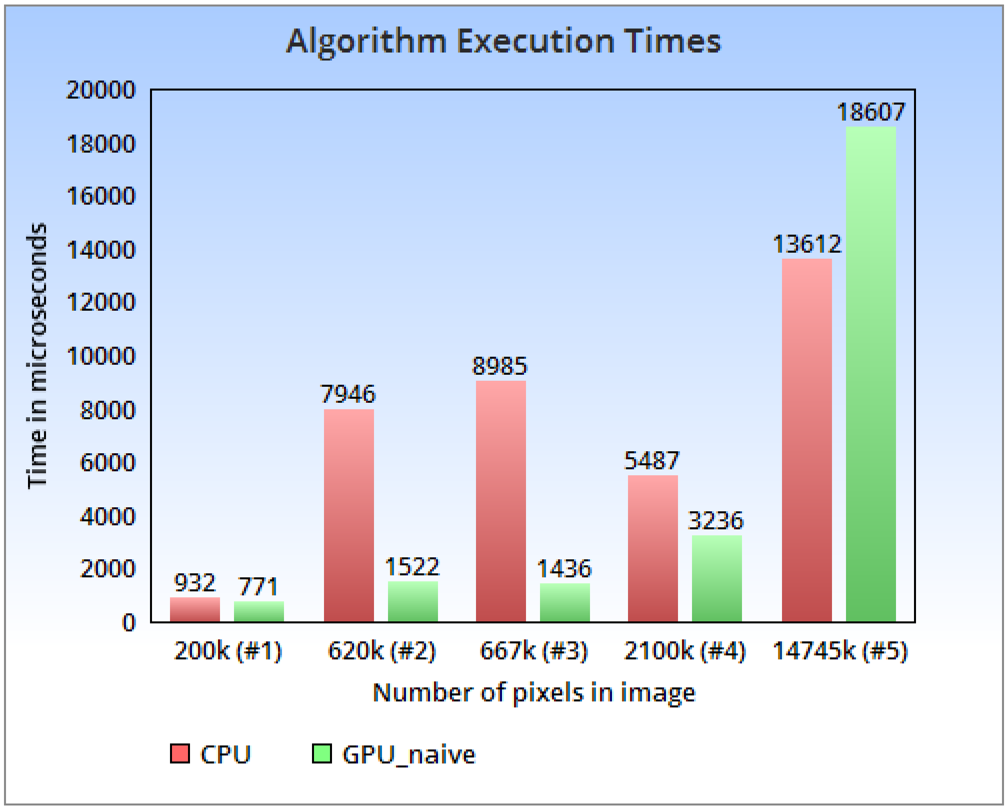
1920x1080

Image 5

5120x2880

Image number	CPU total time (µs)	Time to transfer image from CPU to GPU (µs)	The time to equalize the histogram on the GPU for the input image (µs)	The time to transfer the equalized image back to CPU (µs)	GPU total time (µs)
1	932	112	563	96	771
2	7946	223	996	303	1522
3	8985	216	1011	208	1436
4	5487	563	2196	476	3236
5	13612	3299	11958	3349	18607

*Time is measured in microseconds (Config 1)



Time analysis for CPU and naive GPU implementation (config 1)

Device specifications for measurements:

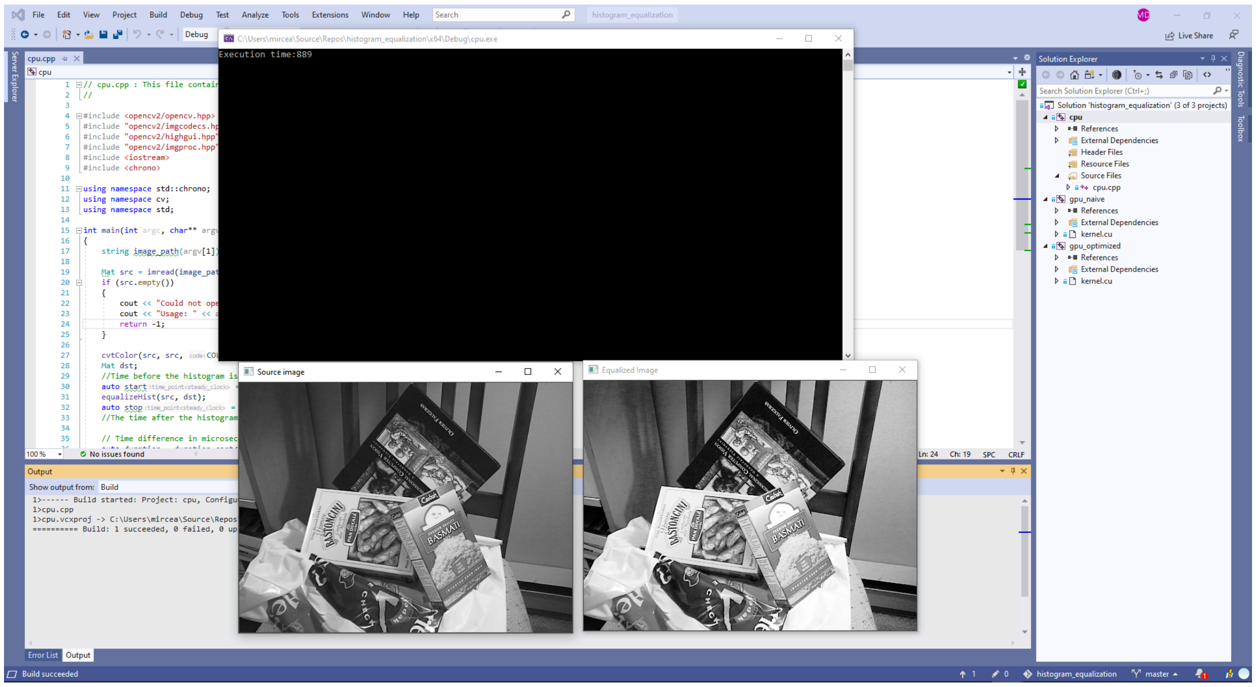
Chelaru Ștefan:

- CPU: i7-9750H CPU @ 2.60GHz
- 16Gb RAM
- GPU: Nvidia GTX 1650 (CUDA capability version 6.1, CUDA driver version 11.1) GDDR6
- Visual Studio 2019

Dobreanu Mircea:

- CPU: Ryzen 3 1200 @ 3.8 GHz
- 16Gb RAM
- GPU: Nvidia GTX 1050 (CUDA capability version 6.1, CUDA driver version 11.1) GDDR5
- Visual Studio 2019

Example runs



```
// cpu.cpp : This file contain
//

#include <opencv2/opencv.hpp>
#include "opencv2/imgcodecs.hp
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
#include <chrono>

using namespace std::chrono;
using namespace cv;
using namespace std;

int main(int argc, char** arg
{
    string image_path(argv[1])

    Mat src = imread(image_pat
    if (src.empty())
    {
        cout << "Could not ope
        cout << "Usage: " << a
        return -1;
    }

    cvtColor(src, src, code: CO
    Mat dst;
    //Time before the histogram is
    auto start :time_point<steady_clock> =
    equalizeHist(src, dst);
    auto stop :time_point<steady_clock> =
    //The time after the histogram

    // Time difference in microsec
```

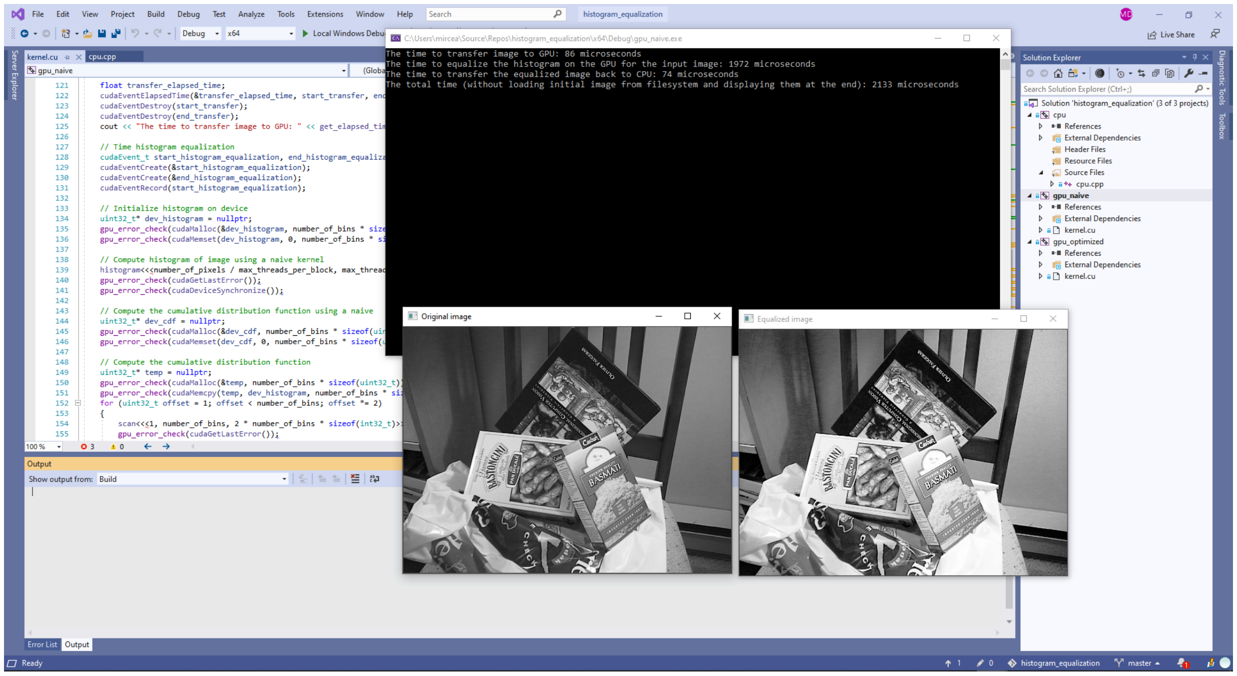
```
C:\Users\mircea\Source\Repos\histogram_equalization\x64\Debug\cpu.exe
Execution time:889
```

Output

Show output from: Build

```
1>------ Build started: Project: cpu, Configu
1>cpu.cpp
1>cpu.vcxproj -> C:\Users\mircea\Source\Repos
========== Build: 1 succeeded, 0 failed, 0 up
```

Example CPU run (First config)



```
float transfer_elapsed_time;
cudaEventElapsedTime(&transfer_elapsed_time, start_transfer, enc
cudaEventDestroy(start_transfer);
cudaEventDestroy(end_transfer);
cout << "The time to transfer image to GPU: " << get_elapsed_tim

// Time histogram equalization
cudaEvent_t start_histogram_equalization, end_histogram_equaliza
cudaEventCreate(&start_histogram_equalization);
cudaEventCreate(&end_histogram_equalization);
cudaEventRecord(start_histogram_equalization);

// Initialize histogram on device
uint32_t* dev_histogram = nullptr;
gpu_error_check(cudaMalloc(&dev_histogram, number_of_bins * size
gpu_error_check(cudaMemset(dev_histogram, 0, number_of_bins * si

// Compute histogram of image using a naive kernel
histogram<<<number_of_pixels / max_threads_per_block, max_thread
gpu_error_check(cudaGetLastError());
gpu_error_check(cudaDeviceSynchronize());

// Compute the cumulative distribution function using a naive
uint32_t* dev_cdf = nullptr;
gpu_error_check(cudaMalloc(&dev_cdf, number_of_bins * sizeof(uir
gpu_error_check(cudaMemset(dev_cdf, 0, number_of_bins * sizeof(u

// Compute the cumulative distribution function
uint32_t* temp = nullptr;
gpu_error_check(cudaMalloc(&temp, number_of_bins * sizeof(uint32_t))
gpu_error_check(cudaMemcpy(temp, dev_histogram, number_of_bins * siz
for (uint32_t offset = 1; offset < number_of_bins; offset *= 2)
{
    scan<<<1, number_of_bins, 2 * number_of_bins * sizeof(int32_t)>>
    gpu_error_check(cudaGetLastError());
```

C:\Users\mircea\Source\Repos\histogram_equalization\x64\Debug\gpu_naive.exe

```
The time to transfer image to GPU: 86 microseconds
The time to equalize the histogram on the GPU for the input image: 1972 microseconds
The time to transfer the equalized image back to CPU: 74 microseconds
The total time (without loading initial image from filesystem and displaying them at the end): 2133 microseconds
```

Example GPU run (first config)

Note: These runs are only for showing the program working over images. These screenshots were not done on the machine used for measuring.

First optimization

While looking at the profiling data from the naive implementation, we discovered that the most demanding kernel is actually the one that reads the pixel's intensity from global memory and stores in another global memory location the modified value for that pixel according to the cumulative distribution function.

Since at least a read from global memory and a write to global memory is required for the core purpose of the function, we've tried caching the cumulative distribution function in each block's shared memory but this provided no improvement benefits.

So for now, we've settled with trying to improve the computation of the histogram kernel by using changing the memory access patterns; we now make use of a block's shared memory and by modifying the call of the kernel. So we went from this kernel:

```
__global__ void histogram(const uint8_t* image, uint32_t* histogram)
{
    const uint32_t index = blockDim.x * blockIdx.x + threadIdx.x;
    atomicAdd(&histogram[image[index]], 1);
}
```

And this call:

```
histogram<<<number_of_pixels / max_threads_per_block, max_threads_per_block>>>(dev_image, dev_histogram);
```

Improved version:

```
__global__ void histogram(const uint8_t* image, uint32_t* histogram, const size_t number_of_pixels)

{

    const uint32_t tid = blockDim.x * blockIdx.x + threadIdx.x;

    // Initialize shared memory of block with all zeroes

    __shared__ uint32_t block_histogram[number_of_bins];

    if (threadIdx.x < number_of_bins)

    {

        block_histogram[threadIdx.x] = 0;

    }

    __syncthreads();

    // Count all the pixel values of the pixels assigned to current block

    for (size_t index = tid; index < number_of_pixels; index += (gridDim.x * blockDim.x))

    {

        atomicAdd(&block_histogram[image[index]], 1);

    }

    __syncthreads();

    // Add the partial appearance counters to the total appearances counters in global memory

    if (threadIdx.x < number_of_bins)

    {

        atomicAdd(&histogram[threadIdx.x], block_histogram[threadIdx.x]);

    }

}
```

With the following call:

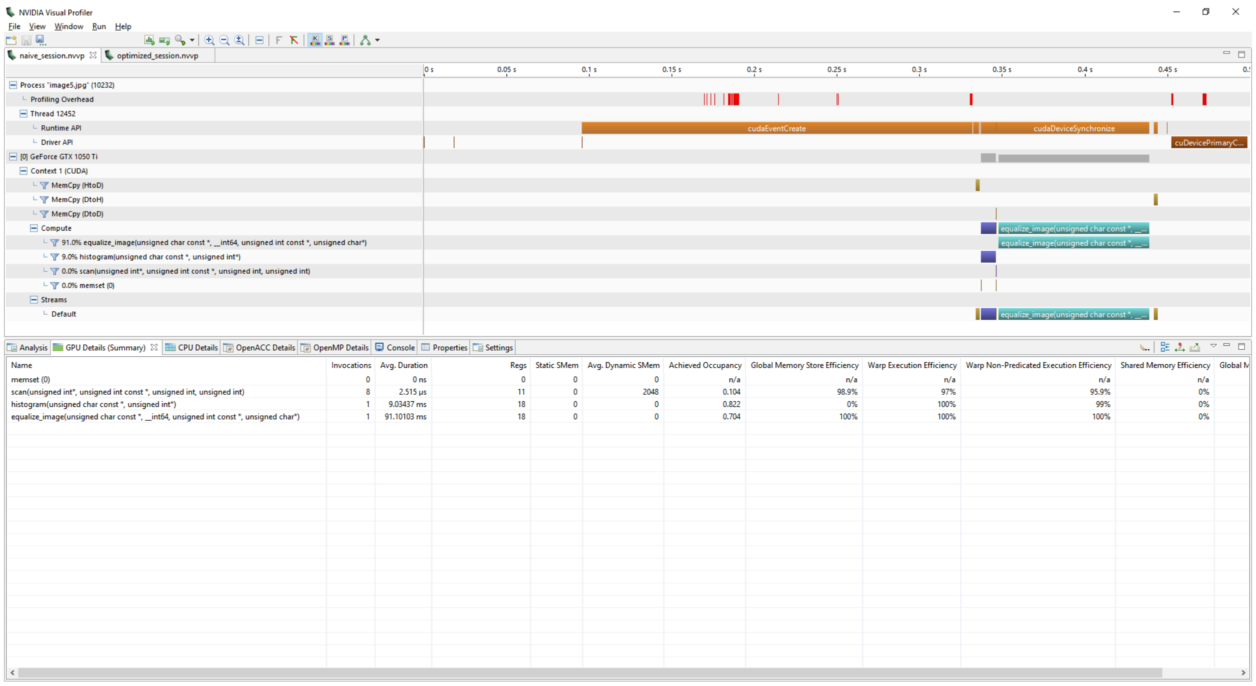
```
size_t histogram_thread_load_factor = 128;

histogram << <number_of_pixels / max_threads_per_block / histogram_thread_load_factor,
max_threads_per_block >> > (dev_image, dev_histogram, number_of_pixels);
```

The improved version, while more complicated, is the same or faster depending on the "load factor". Basically if the load factor is 1, each thread executing the kernel does a read from global memory and an atomic operation on a location in global memory, which is exactly what the original function did. If the load factor is 1, logically it is the same, but it has the same performance (or slightly worse) because you also add reads from shared memory.

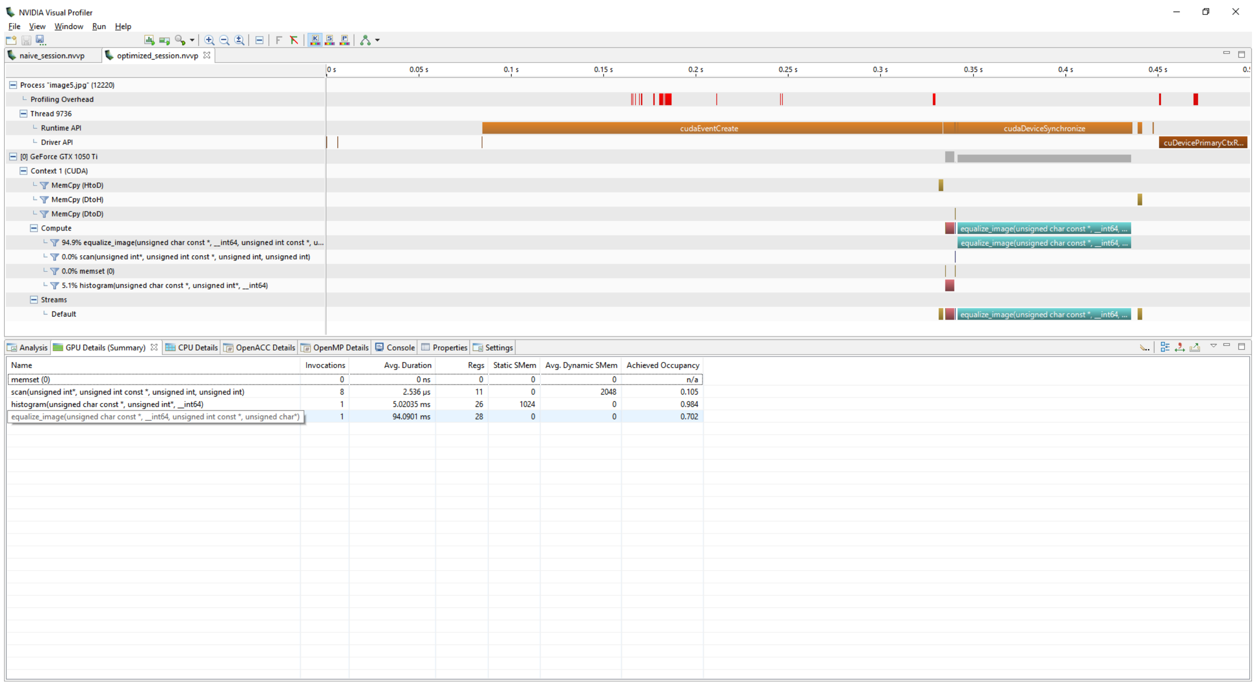
But if you increase the load factor, then each thread is not incrementing a memory position for just one pixel, but for multiple pixels. We've tried this with powers of 2, and the best performance benefit is seen at a load factor of 128 pixels/thread. After that, performance seems to fall off, albeit slightly.

This works because we are computing a sub-histogram for all the pixels assigned to all the threads within a block. We are replacing global atomic calls with faster shared memory atomic calls, which gives us a performance boost. We've observed an improvement in Visual Profiler from 9 milliseconds in the naive version to 5 milliseconds for the largest image we have (5120x2880); this is an 1.8x improvement on the kernel. See the next page for screenshots of the profiling data.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Achieved Occupancy	Global Memory Store Efficiency	Warp Execution Efficiency	Warp Non-Predicated Execution Efficiency	Shared Memory Efficiency	Global M
memset (0)	0	0 ns	0	0	0	n/a	n/a	n/a	n/a	n/a	
scan(unsigned int*, unsigned int const *, unsigned int, unsigned int)	8	2.515 μs	11	0	2048	0.104	98.9%	97%	95.9%	0%	
histogram(unsigned char const *, unsigned int*)	1	9.03437 ms	18	0	0	0.822	0%	100%	99%	0%	
equalize_image(unsigned char const *, __int64, unsigned int const *, unsigned char*)	1	91.10103 ms	18	0	0	0.704	100%	100%	100%	0%	

Naive run



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Achieved Occupancy
memset (0)	0	0 ns	0	0	0	n/a
scan(unsigned int*, unsigned int const *, unsigned int, unsigned int)	8	2.536 μs	11	0	2048	0.105
histogram(unsigned char const *, unsigned int*, __int64)	1	5.02035 ms	26	1024	0	0.984
equalize_image(unsigned char const *, __int64, unsigned int const *, unsigned char*)	1	94.0901 ms	28	0	0	0.702

Improved run

Second optimization

For the second optimization we managed to bring down the execution time of the equalization kernel (the one which evaluates all the new values of the image). We were trying to use fewer threads and offset some of the cost of starting many threads by making them longer lived; so we tried giving each thread.

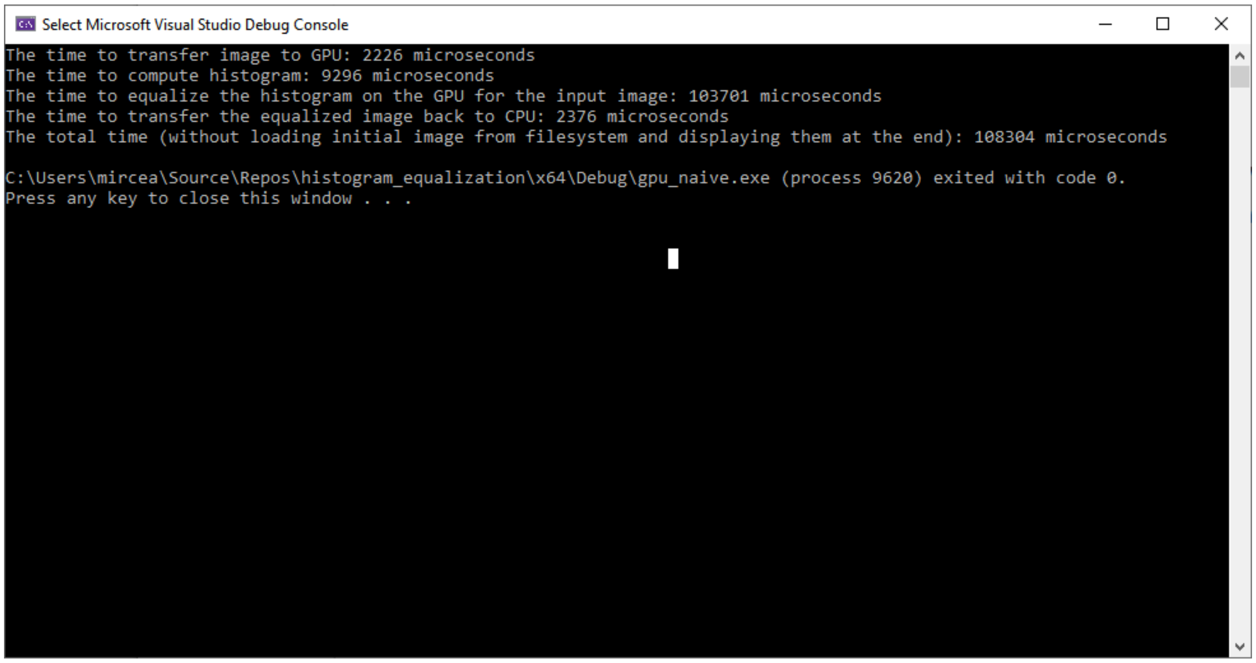
So by giving each thread more work we would have a for loop which calculated its load (let's say 32 pixels per each thread). We moved the factor of (number_of_bins -1) / (number_of_pixels- 1) outside the for loop since it would be recomputed each time. We were also more explicit with our intentions, so the compiler could better optimize our code. What we ended up with is this:

```
const int number_of_bins = 256;
__global__ void equalize_image(const uint8_t* original_image,
  const size_t number_of_pixels,
  const uint32_t* cdf,
  uint8_t* equalized_image
)
{
  const size_t thread_id = blockDim.x * blockIdx.x + threadIdx.x;
  const float factor = 1.0f * (number_of_bins - 1) / number_of_pixels;

  for (size_t index = thread_id; index < number_of_pixels; index += (gridDim.x * blockDim.x))
  {
    equalized_image[index] = (uint8_t)(cdf[original_image[index]] * factor);
  }
}
```

We then went back and tested further with different invocations; we changed the load per thread to small powers of 2 and so no real difference between running with 1 pixel load per thread versus the others.

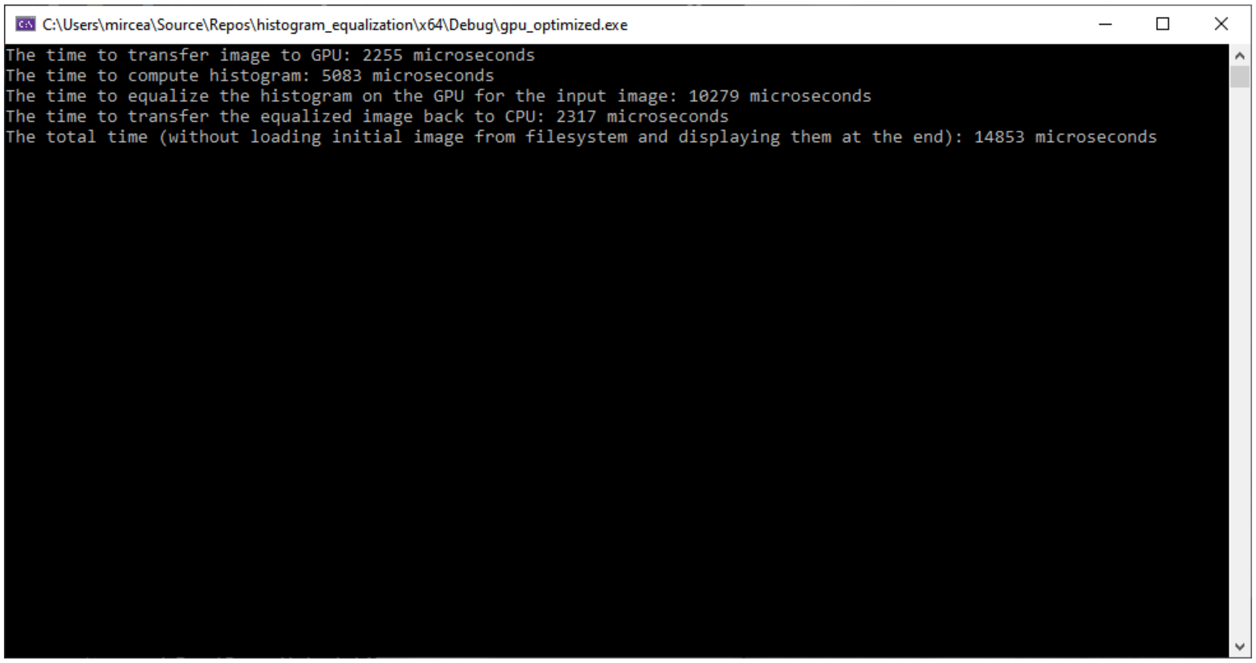
We also analyzed the machine code on Godbolt tool to compare the machine code for the two implementation and you can see that we substituted integer divisions for floating point divisions, and the actual assignment lines produce more machine code in the naive version as opposed to the optimized one.



```
The time to transfer image to GPU: 2226 microseconds
The time to compute histogram: 9296 microseconds
The time to equalize the histogram on the GPU for the input image: 103701 microseconds
The time to transfer the equalized image back to CPU: 2376 microseconds
The total time (without loading initial image from filesystem and displaying them at the end): 108304 microseconds

C:\Users\mircea\Source\Repos\histogram_equalization\x64\Debug\gpu_naive.exe (process 9620) exited with code 0.
Press any key to close this window . . .
```

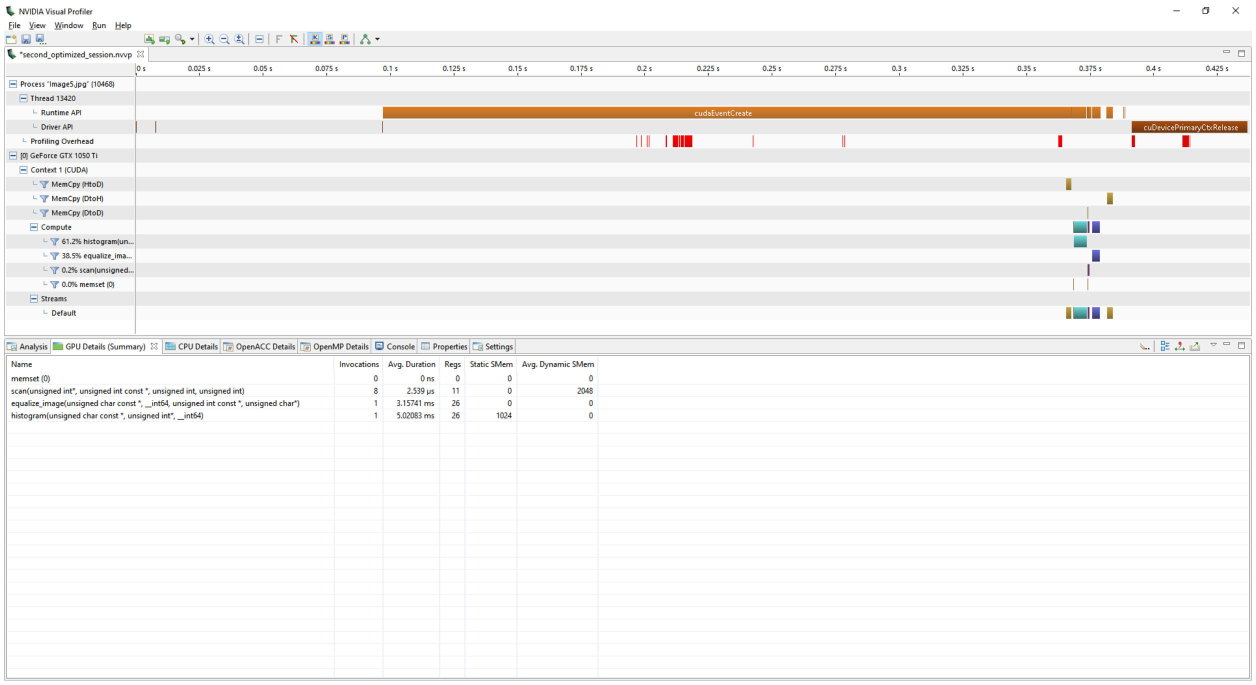
Naive CUDA implementation (second config)


C:\Users\mircea\Source\Repos\histogram_equalization\x64\Debug\gpu_optimized.exe

```
The time to transfer image to GPU: 2255 microseconds
The time to compute histogram: 5083 microseconds
The time to equalize the histogram on the GPU for the input image: 10279 microseconds
The time to transfer the equalized image back to CPU: 2317 microseconds
The total time (without loading initial image from filesystem and displaying them at the end): 14853 microseconds
```

Implementation with the 2 optimizations (second config)

From the above screenshots, you can see that on the second configuration we reduced the execution time from the original 108 ms to just about 15 ms.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
scan(unsigned int*, unsigned int const *, unsigned int, unsigned int)	8	2.539 μs	11	0	2048
equalize_image(unsigned char const *, __int64, unsigned int const *, unsigned char*)	1	3.15741 ms	26	0	0
histogram(unsigned char const *, unsigned int*, __int64)	1	5.02083 ms	26	1024	0

Visual Profiler run for optimized

Conclusions

1. Comparison of times (config 2):
   - CPU: 22ms
   - Naive: 110ms
   - Optimized histogram and equalization kernels: 15 ms
2. We guess that the ideal CUDA implementation could be somewhere like 4 times faster which means there is some optimization that could still be done on our implementation.
3. We expected that the optimizations would make the code maybe harder to understand and also for it to be slightly better than the CPU version which uses the highly optimized OpenCV library. Since the program only handles 1 image and our optimized version runs in ~70% of the time of the CPU version, we can say that it matched our expectations quite well.
4. The naive GPU implementation one is the slowest and it matches our expectations.