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

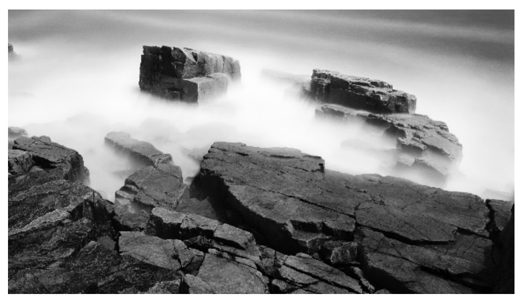
Image 2

910x682

Image 3

1000x667

Image 4



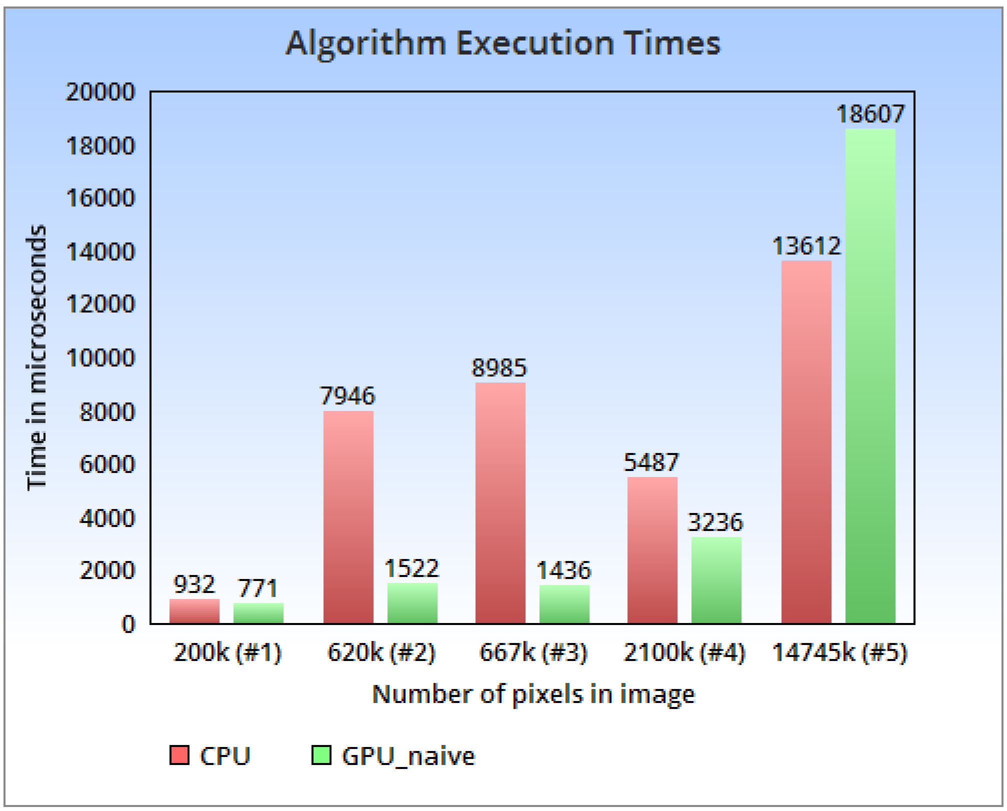
1920x1080

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



Device specifications for measurements:

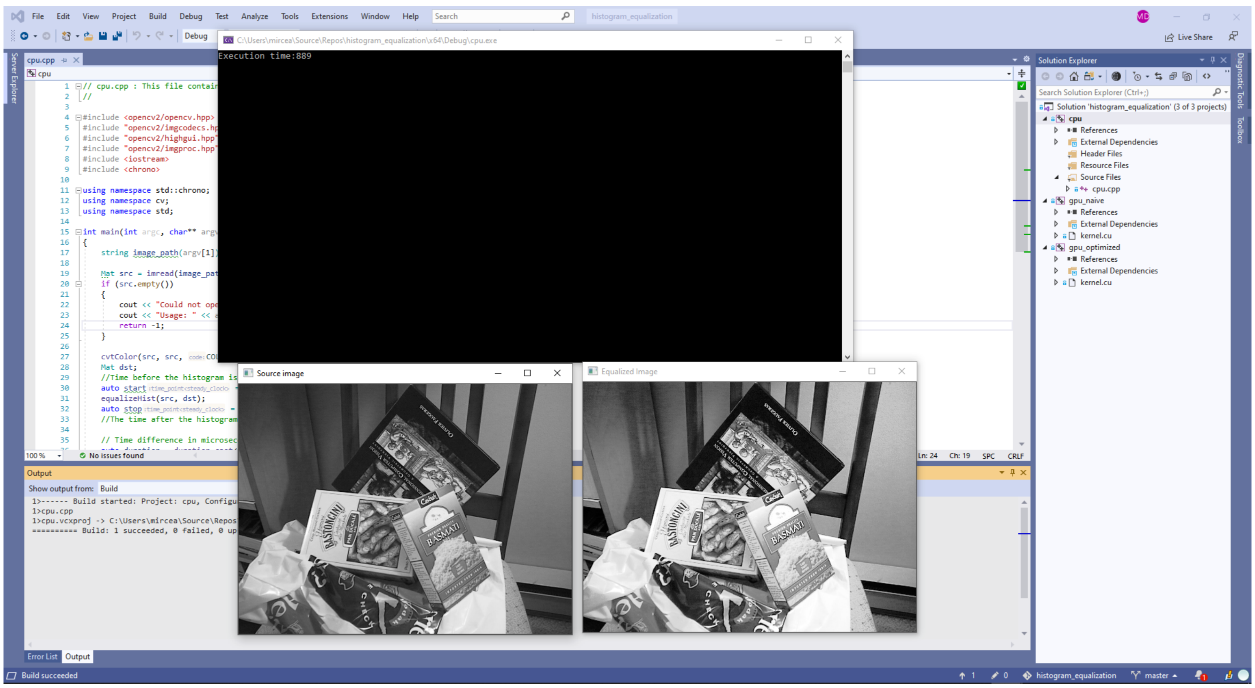
CPU: i7-9750H CPU @ 2.60GHz

16Gb RAM

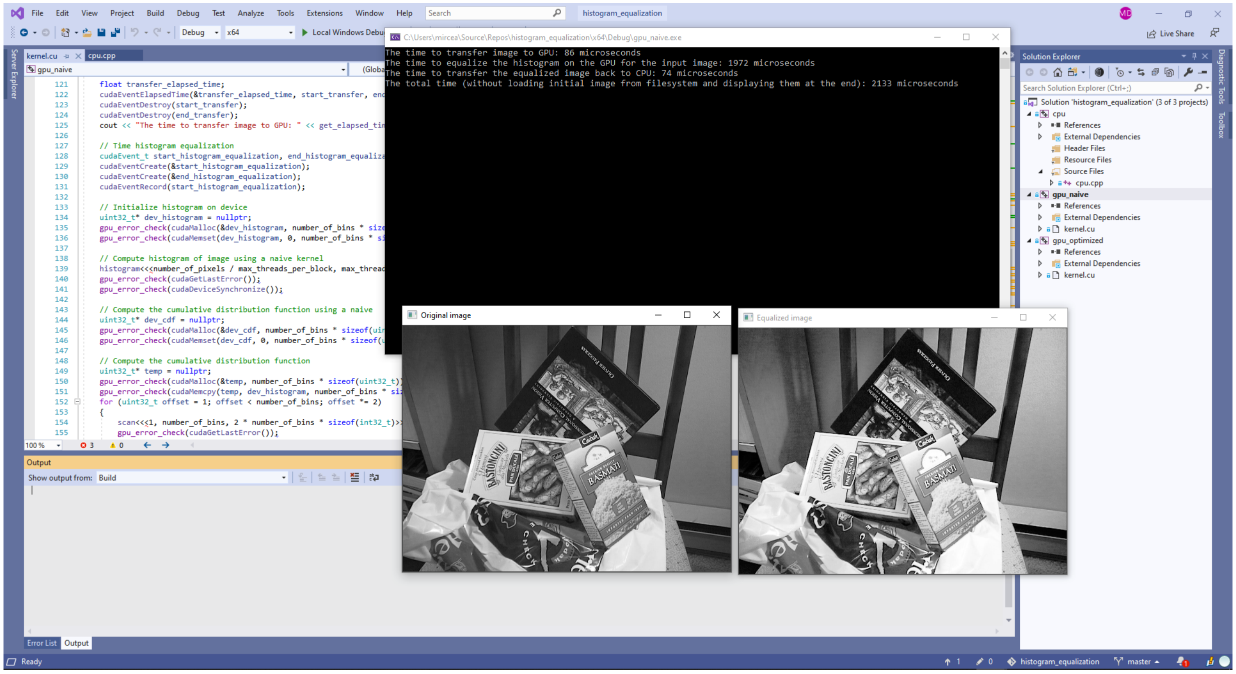
GPU: Nvidia GTX 1650 (CUDA capability version 6.1, CUDA driver version 11.1)

Visual Studio 2019

Example runs



Example CPU run



Example GPU run

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:

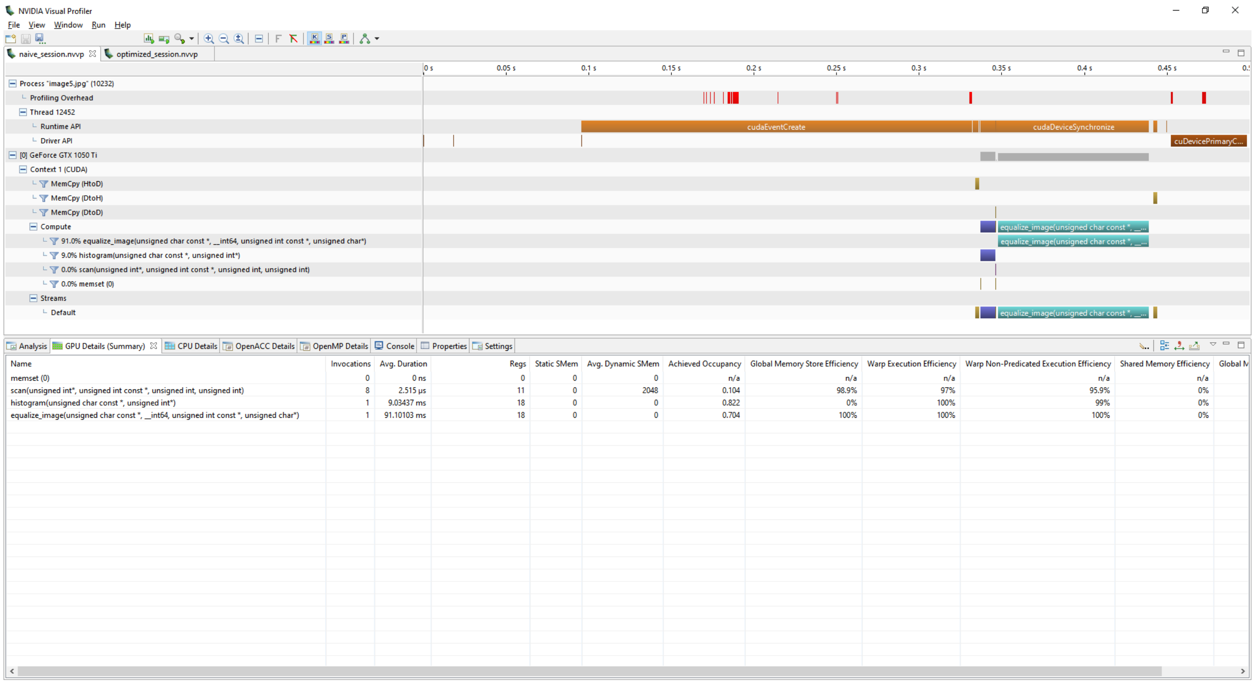
```
size_t histogram_thread_load_factor = 128;

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

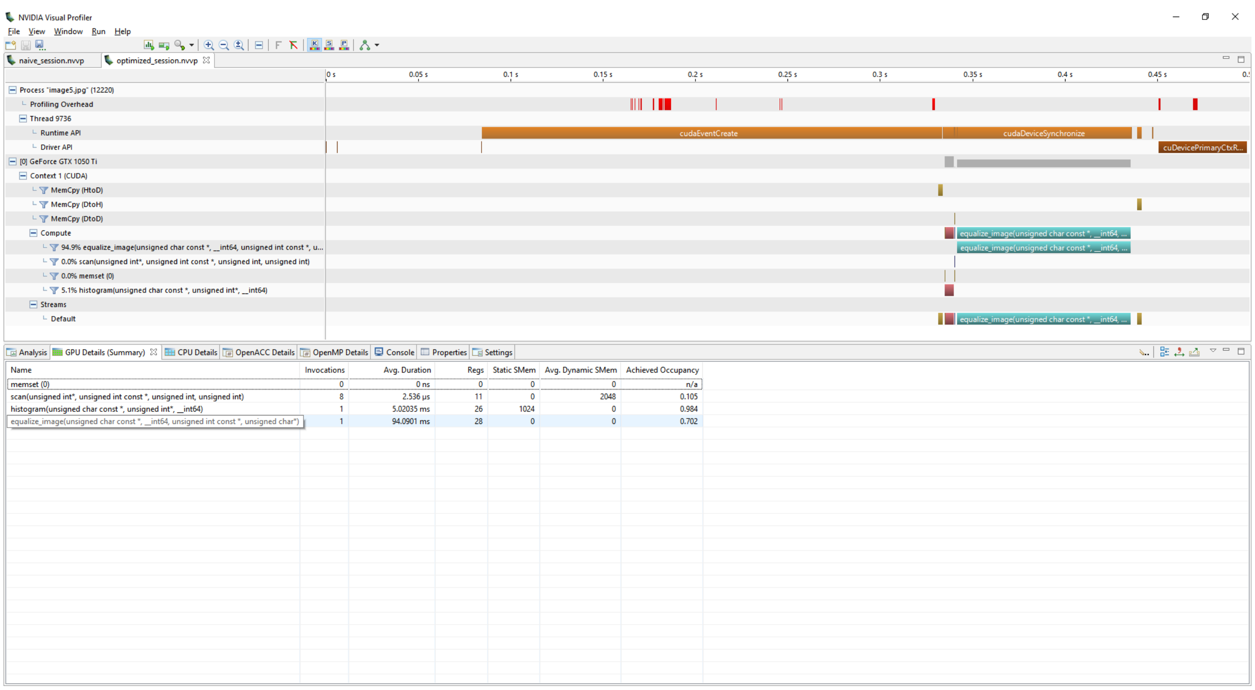
The improved version, while more complicated, is the same or faster depending on the "load factor". Basically if the load factor is 1, each thread executing the kernel does a read from global memory and an atomic operation on a location in global memory, which is exactly what the original function did. 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.



Naive run



Improved run