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 Image 2

512x384 910x682

Image 3 Image 4

1000x667 1920x1080

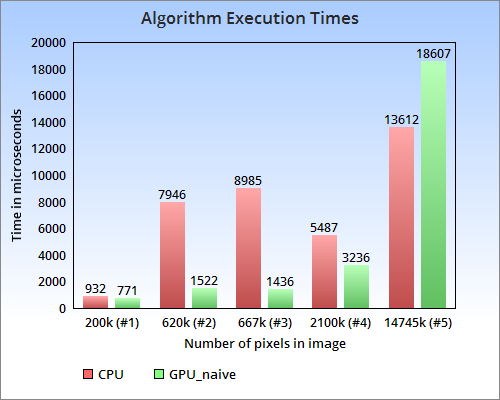
Image 5



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:

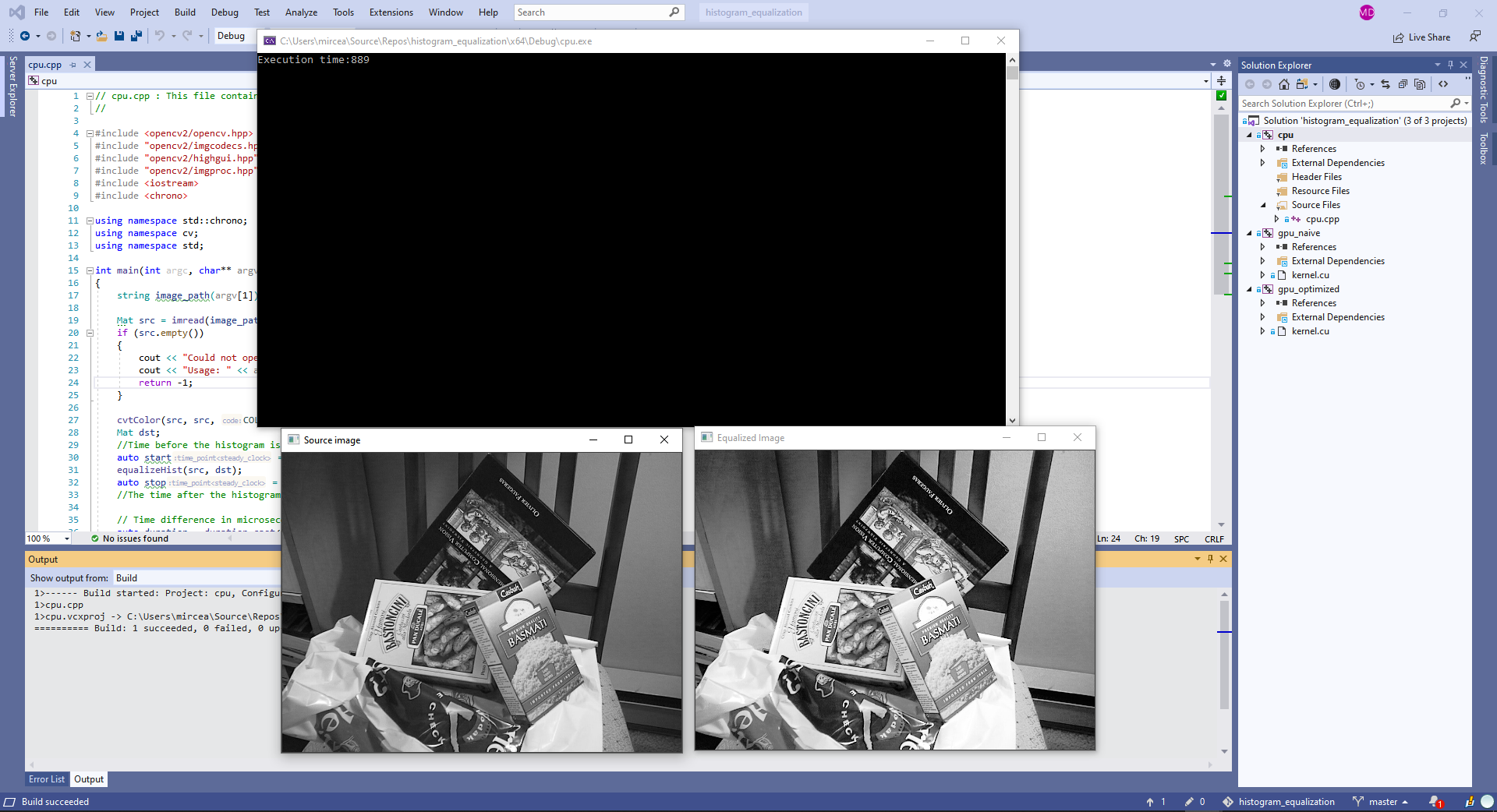
Chelaru Ștefan:

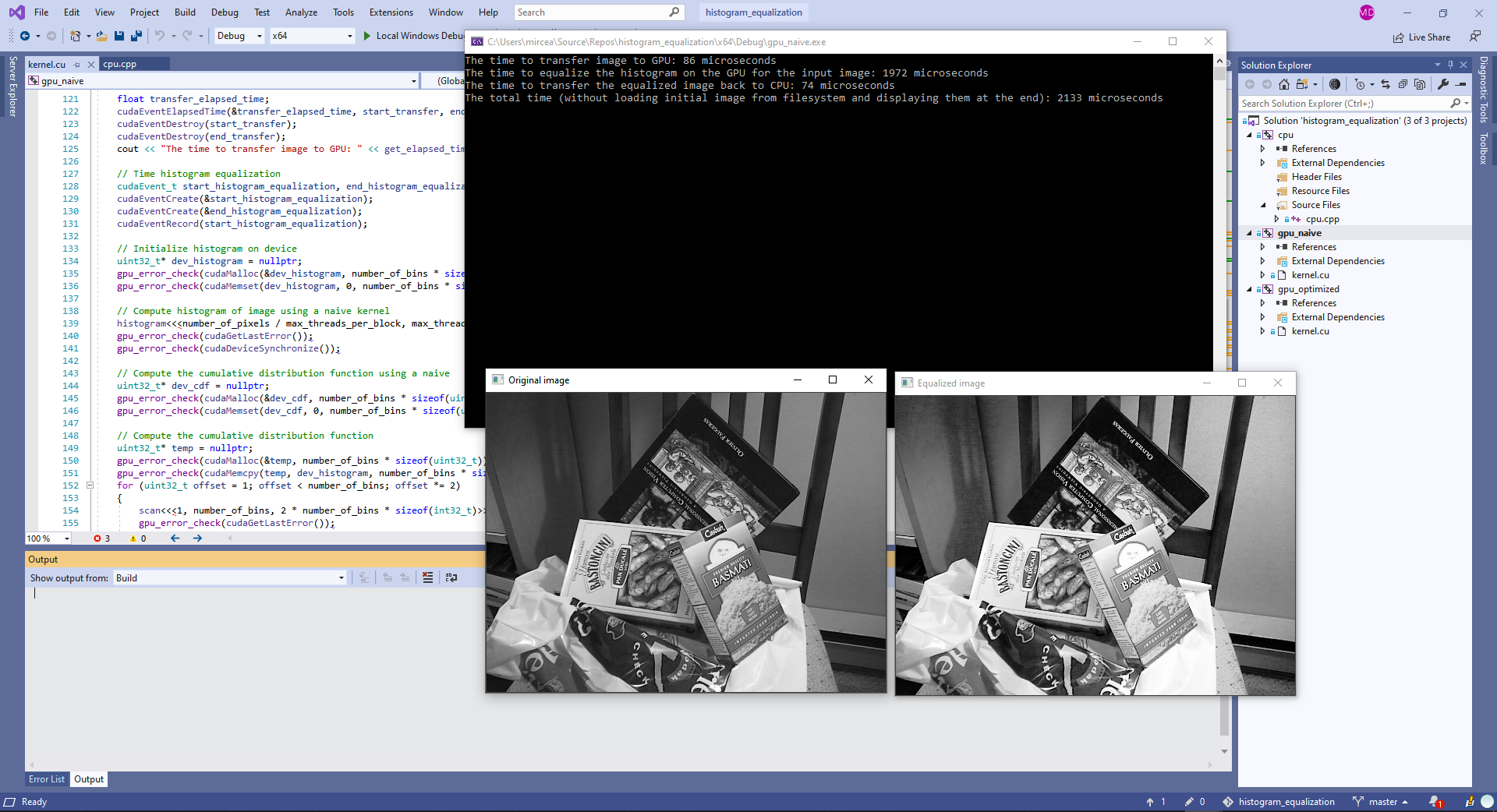
* CPU: i7-9750H CPU @ 2.60GHz
* 16Gb RAM
* GPU: Nvidia GTX 1650 (CUDA capability version 6.1, CUDA driver version 11.1) GDDR6
* Visual Studio 2019

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

Example CPU run (First config)

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:

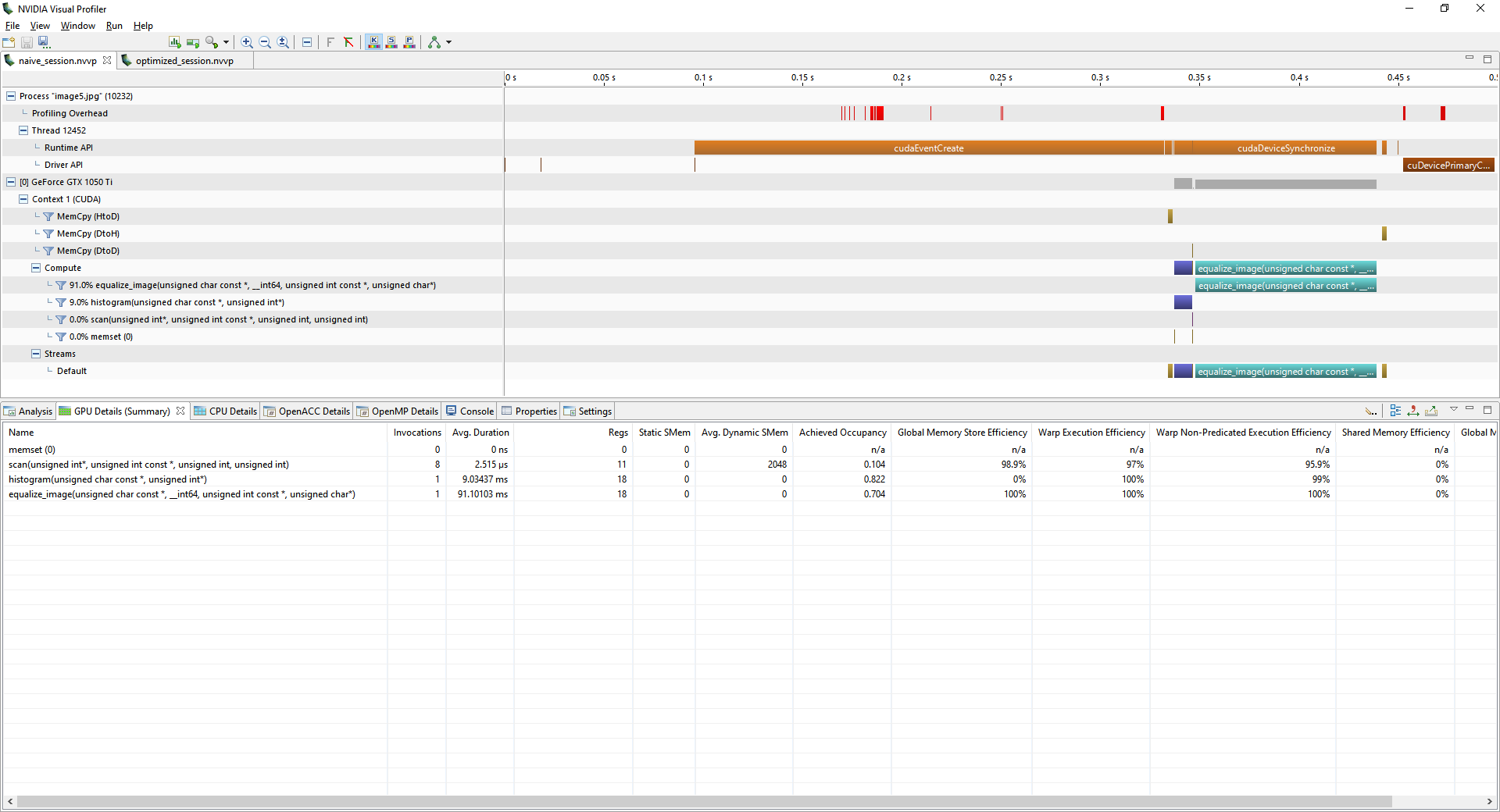
size\_t histogram\_thread\_load\_factor = 128;

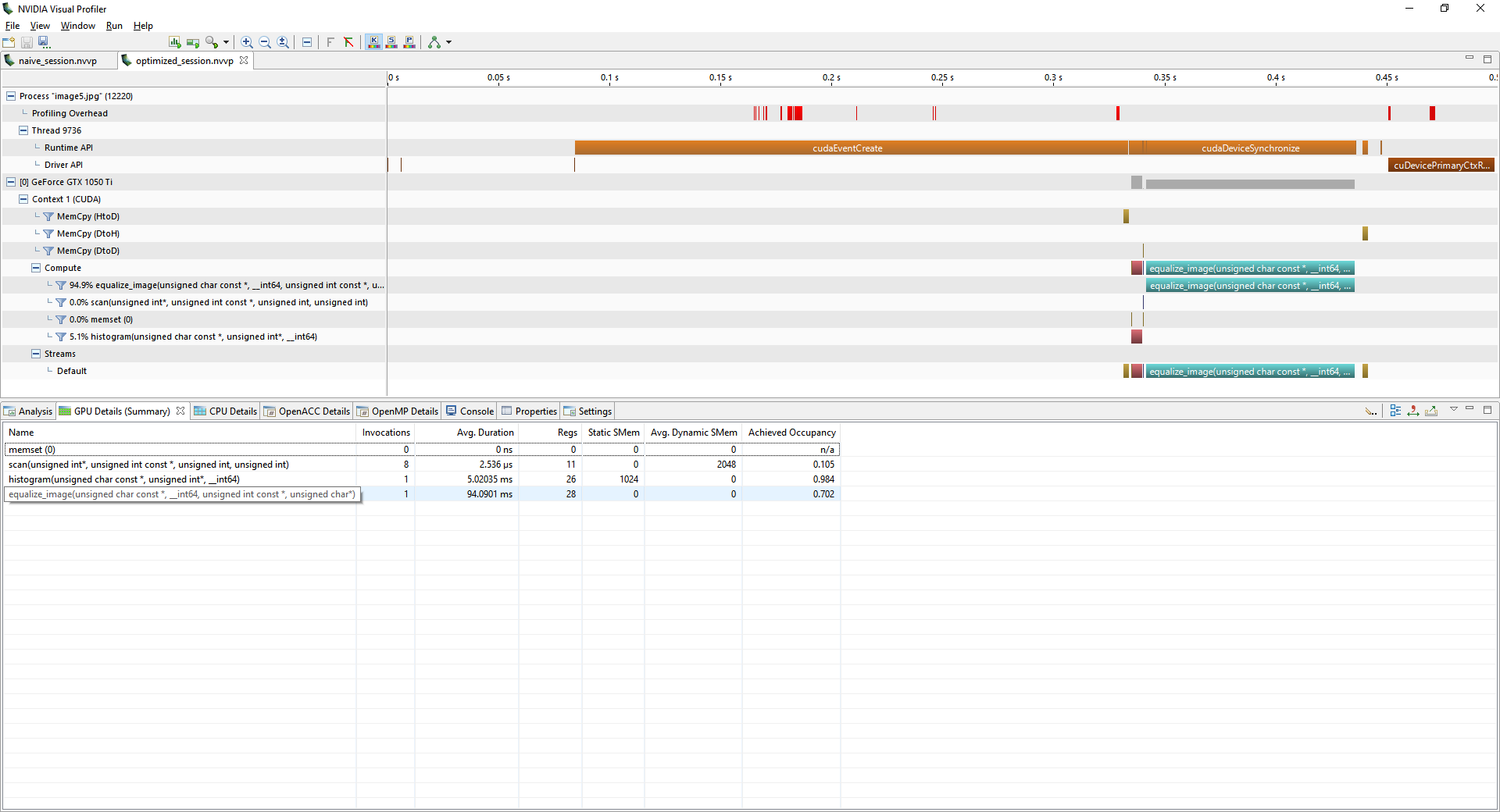
histogram << <number\_of\_pixels / max\_threads\_per\_block / histogram\_thread\_load\_factor, max\_threads\_per\_block >> > (dev\_image, dev\_histogram, number\_of\_pixels);

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.

Naive run

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))

    {

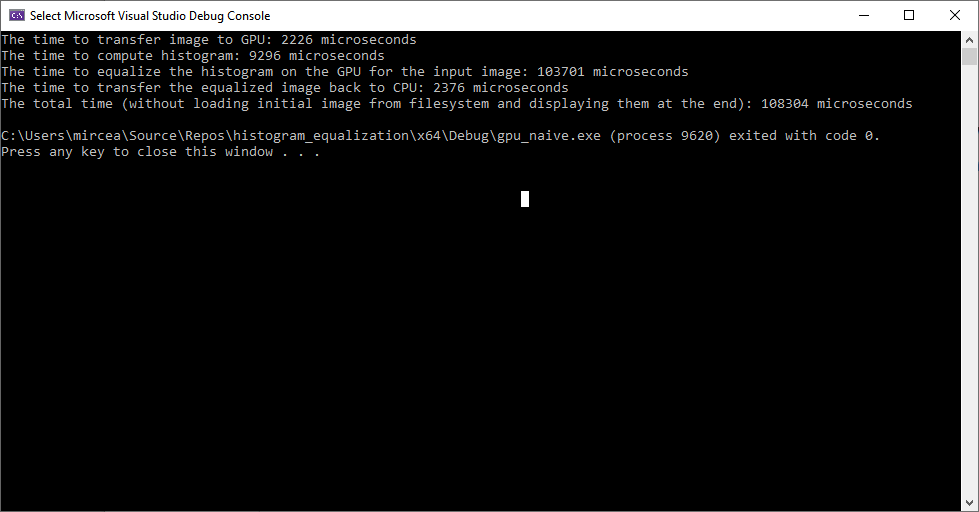
        equalized\_image[index] = (uint8\_t)(cdf[original\_image[index]] \* factor);

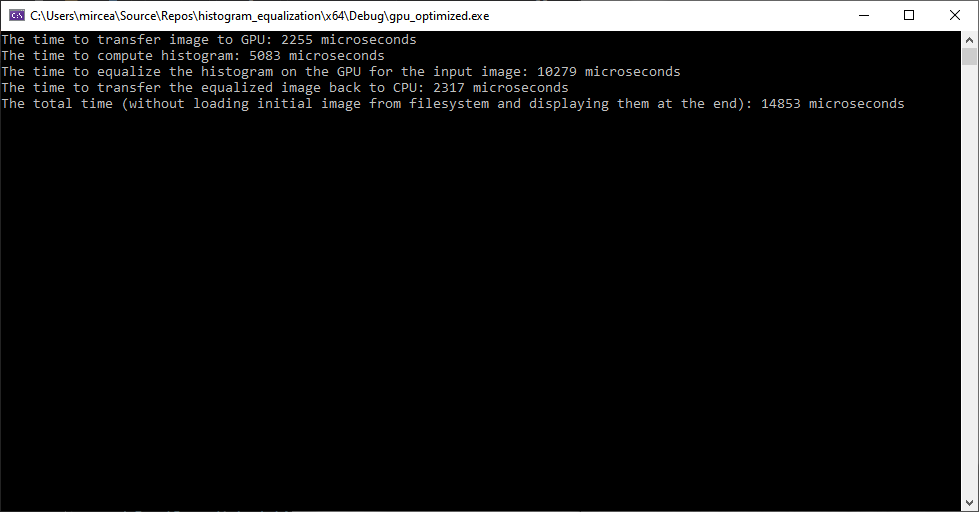
    }

}

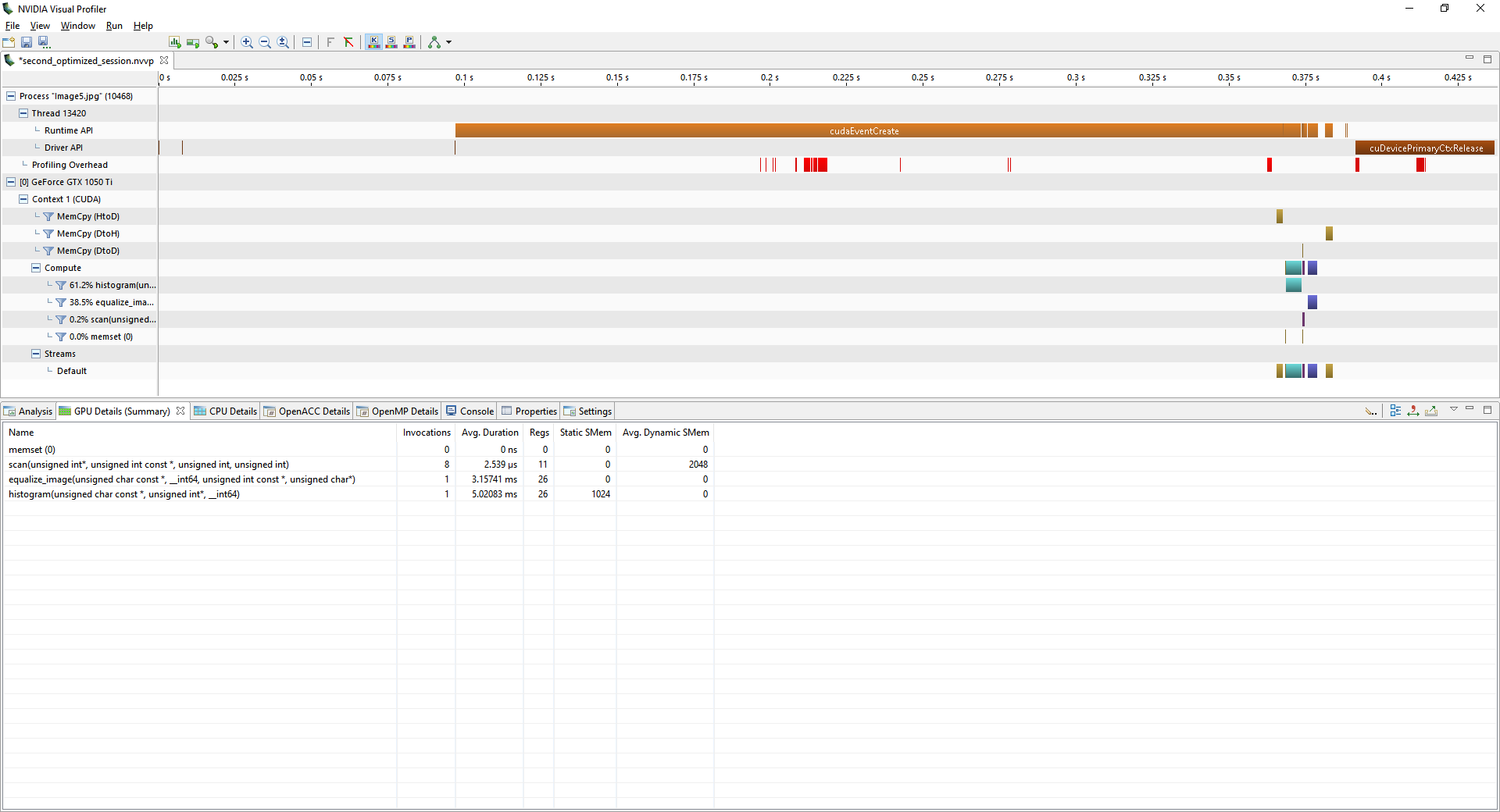
We then went back and tested further with different invocations; we changed the load per thread to small powers of 2 and so no real difference between running with 1 pixel load per thread versus the others.

We also analyzed the machine code on [Godbolt tool](https://godbolt.org/z/zrWKse) 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.

Naive CUDA implementation (second config)

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.

Visual Profiler run for optimized

Conclusions

1. Comparison of times (config 2):

* CPU: 22ms
* Naive: 110ms
* Optimized histogram and equalization kernels: 15 ms

1. 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.
2. 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.
3. The naive GPU implementation one is the slowest and it matches our expectations.