General Purpose GPU Programming

Project

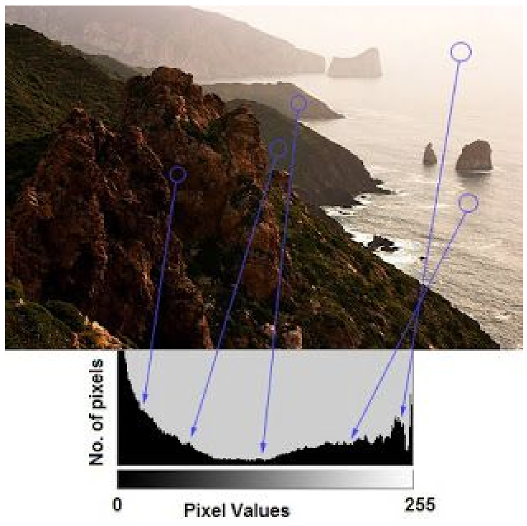
Histogram equalization

Description

What is a histogram?

You can consider a histogram as a graph or plot, which gives you an overall idea about the intensity distribution of an image. It is a plot with pixel values (ranging from 0 to 255) in the X-axis and the corresponding number of pixels in the image on the Y-axis.

It is just another way of understanding the image. By looking at the histogram of an image, you get an intuition about the contrast, brightness, intensity distribution, etc of that image. Almost all image processing tools today provide features on histogram [1].



You can see the image and its histogram on the side. The histogram is drawn for a grayscale image. It can be done for a color image by splitting the image into the RGB components and creating the histogram for each one (R, G, B). The left region of the histogram shows the amount of darker pixels in the image and the right region shows the amount of brighter pixels. From the histogram, you can see the dark region is more than the brighter region, and the number of mid-tones has fewer pixels [1].

Histogram Equalization is an image processing technique that adjusts the contrast of an image by using its histogram. To enhance the contrast of the image, it spreads out the most frequent pixel intensity values or stretches out the intensity range of the image. By accomplishing this, histogram equalization allows the areas with lower contrast to gain a higher contrast [2].

Histogram Equalization can be used when you have images that look washed out because they do not have enough contrast. In such photographs, the light and dark areas blend together creating a flatter image that lacks highlights and shadows [2].

Solution

1. Sequential approach

To compute the histogram the image is iterated similar to a matrix and each value is counted. The output array has a size of 256 (exactly the values that a gray image can have) and it contains values up to image height multiplied by image width. Finally, in order to display a histogram, the values are normalized to fit a fixed-sized image that is not abnormally tall.

For the equalizing task, the cumulative frequency of all values is computed and divided by the total number of pixels and multiply them by the maximum gray count (pixel value) in the image.

2. Parallel approach

The operations are computations on arrays. The histogram arrays have the length 256 (the size of the histogram) and the image array has image width multiplied by image height size. Two types of kernels are used with different input vectors and output vector lengths: 256 and image size. For example in histogram calculation, each core takes a single value from the image and increments it with 1 to the previous corresponding position in the frequency array. Because the threads try to read and write the same piece of memory at the same time, an atomic add (a blocking operation) is used to solve the problem.

2.1. Parallel approach - naive

The algorithm for computing the image histogram contains two tasks: computing the histogram and equalizing the histogram.

To compute the histogram we iterated the image array (converted the matrix image into an array) and counted each value. The output array has a size of 256 and it contains values up to image height multiplied by image width. Finally, in order to display a histogram, we had to normalize the values to fit a fixed-sized image that is not abnormally tall.

For using the CUDA device, the memory is allocated to the host (CPU) and device (GPU). The memory needed is copied from the host to the device. The kernels are then called and the processing is done on the GPU. After the GPU kernel is finalized, a synchronization step is necessary. The memory is then copied from the device to the host to be available to the user. The final step is deallocating the memory from the host and device.

2.3. Parallel approach - shared memory

The algorithm is the same as the naive version, with modification to the implementation. The memory is allocated globally (accessible for both host and device). The simple kernels that only have one operation applied to arrays only use global memory.

Shared memory is used for the histogram kernel and cumulative histogram kernel. The histogram kernel [3] computes the partial histogram in every block and then contributes to the global histogram. Each block contains 256 threads, the number of elements in the histogram array. Atomic add operations are used for synchronization. The cumulative histogram is a prefix sum or a scan. Hillis & Steele Parallel Scan algorithm [4] is used for computing the cumulative histogram, shown in the pseudocode below:

```
parallel_scan(x, M):
    for $d$ := 0 to M-1 do
        forall $k$ in parallel do
            if $k - 2^d \geq 0$ then x[out][ k] := x[in][ k] + x[in][ k − $2^d$]
            else x[out][ k] := x[in][ k]
        endforall
        swap(in, out)
    endfor
```

2.4. Parallel approach - GPU optimizations

The first optimization that can be done is running multiple kernels in parallel if possible. Until this stage, all kernels run sequentially and a synchronization step is performed after each one. The **prkKernel**, **skKernel,** and **pskKernel** kernels can be run in parallel and after them, synchronization is needed. The final two kernels **finalValuesKernel** and **finalImageKernel can** also be run in parallel.

Changed the kernel for computing the Histogram. It computes the private histogram for each block and then combines it at the end. It uses blocks with 1024 threads.

Added an optimized version of the kernel for computing the cumulative histogram. The parallel algorithm pattern is *Balanced Trees* [4]. It builds a balanced binary tree on the input data and sweeps it to and then from the root. The tree is not an actual data structure, but a concept to determine what each thread does at each step.

For the cumulative histogram (scan) it traverses down from leaves to root building partial sums at internal nodes in the tree. The root holds the sum of all leaves. This is a reduction algorithm. Traverse back up the tree building the scan from the partial sums. The algorithm is presented below:

```
parallel_scan(x, M):
    for k = 0 to M-1
        offset = $2^k$
        for j = 1 to $2^{M-k-1}$ in parallel do
            $x[j·2^{k+1}-1] = x[j·2^{k+1}-1] + x[j·2^{k+1}-2^k-1]$
        endfor
    endfor
    for k = M-1 to 0
        offset = $2^k$
        for j = 1 to 2M-k-1 in parallel do
            dummy = x[j·2k+1-2 k-1]
            x[j·2k+1-2 k-1] = x[j·2k+1-1]
            x[j·2k+1-1] = x[j·2k+1-1] + dummy
        endfor
    endfor
```

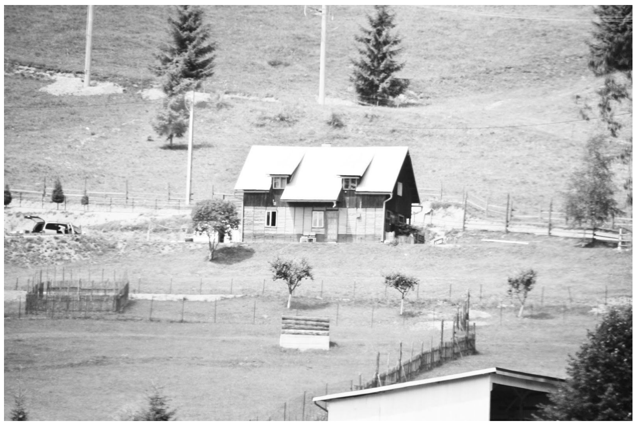
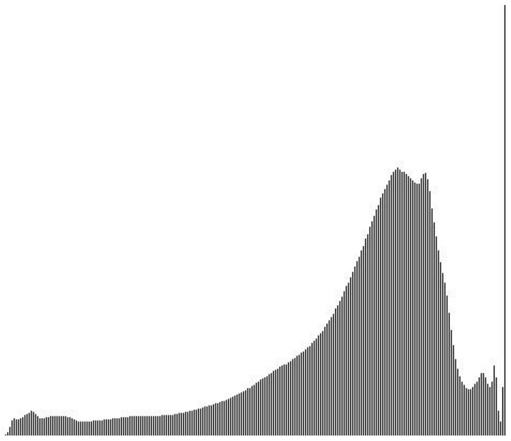
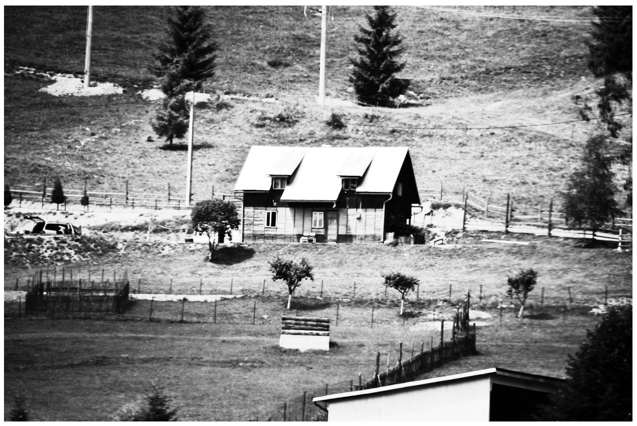
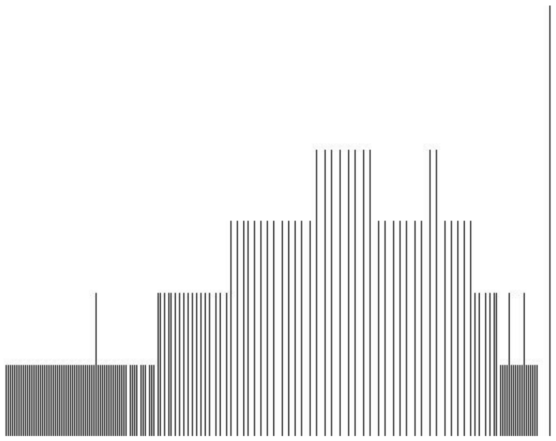
Resources

A device with the following hardware and software components is used for experiments:

- **CPU:** I7-8750
- **GPU:** GTX 1060
- **Operating System:** Windows 10
- **OpenCV version:** 4.1.2
- **CUDA version:** 11.1
- **IDE:** Visual Studio 2019

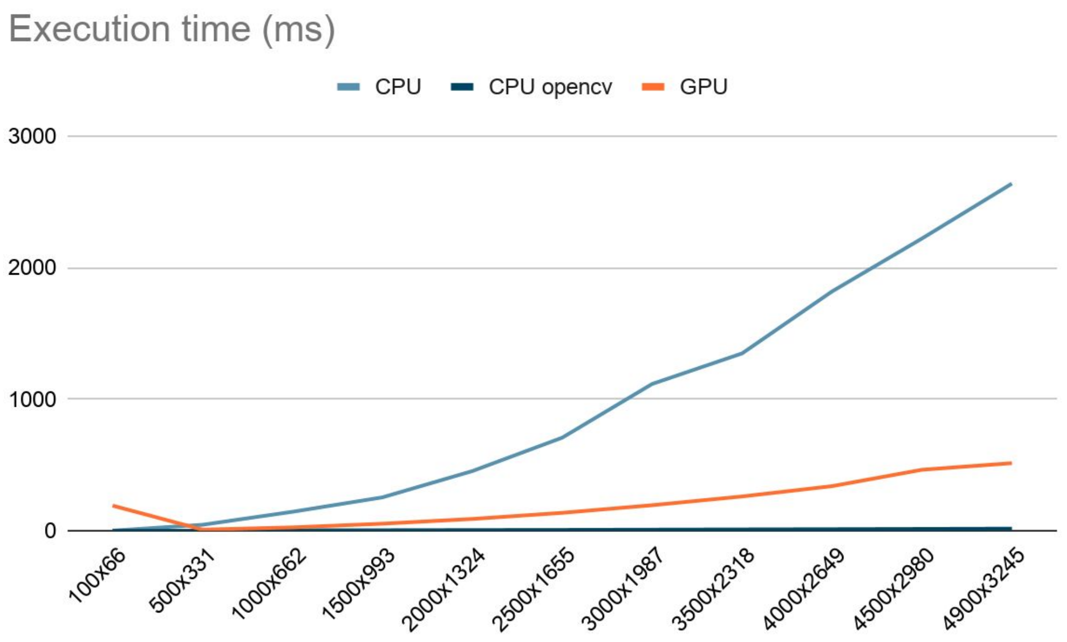
Results

The image used for testing is presented in the following table. The first row shows the original image taken from a DSLR camera. The image is monochrome (grayscale) and is underexposed with low contrast. The histogram is on the right side of the image and it is centered on a specific value. The second row contains the image with the equalized histogram. The histogram shows values that are spread out on the whole range. The resulting image has better contrast.

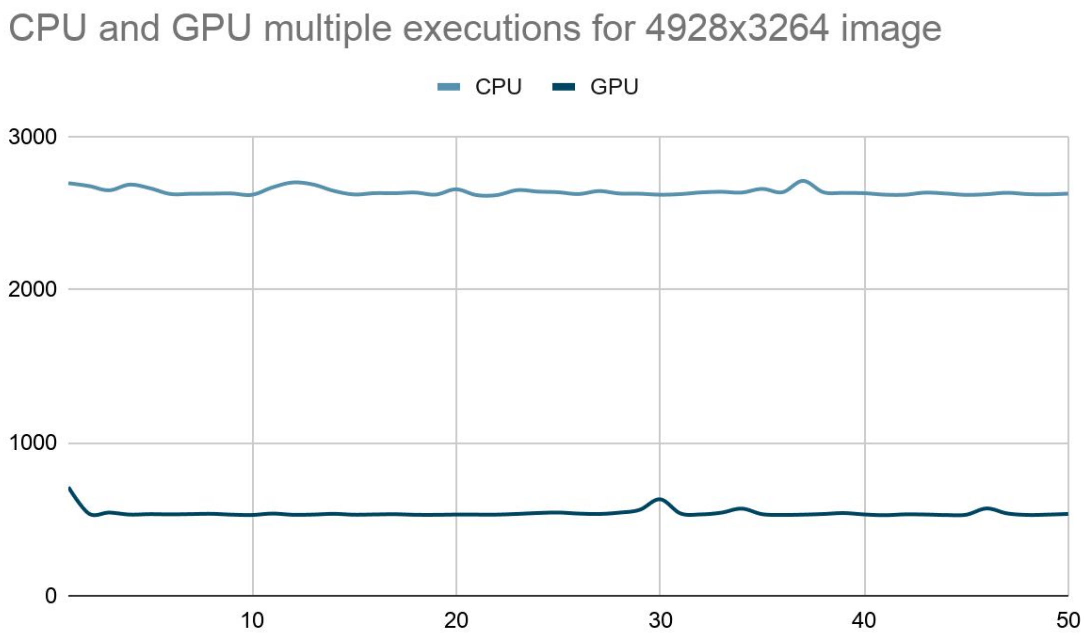
Image	Histogram
	
	

1. First experiment - comparing CPU and GPU implementation

The execution time is presented in the following chart. The lines represent the execution time (in ms) on the CPU single-threaded, CPU with implementation from OpenCV, and GPU with CUDA. The dataset contains the same image, with different resolutions, from 100x36 to 4900x3245. For the sizes under 1000x662, the CPU implementations offer a smaller execution time because the GPU version loses time on memory allocations and transfers from CPU to GPU and vice-versa. From 1000x662, the GPU offers a slower execution time.

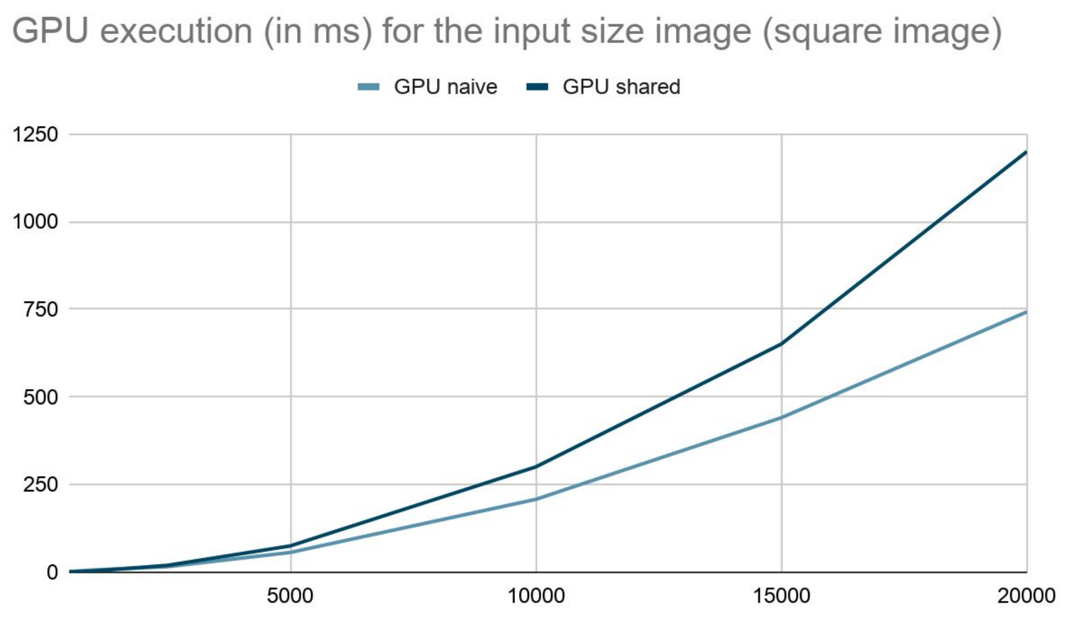


Multiple executions of the same implementation can have different execution times because of the OS task manager. Below can be seen the variation of the times on 50 executions of the CPU and the GPU versions for an image with size 4928x3284.

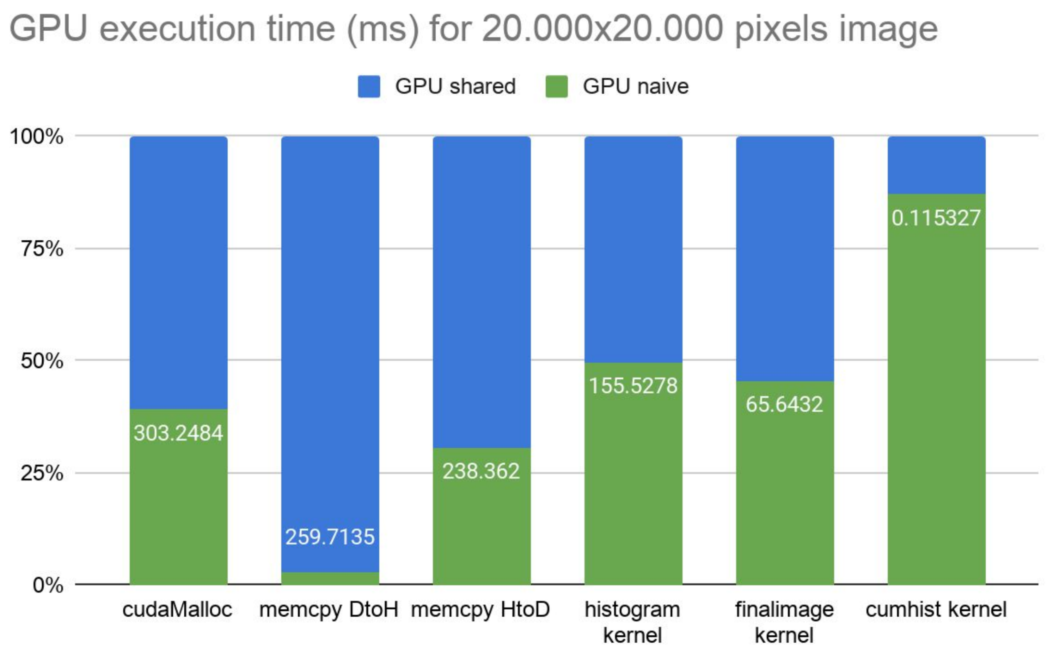


2. Second experiment - improving the GPU implementation with shared memory

The execution time comparison between the naive and shared memory CUDA implementations is shown below:



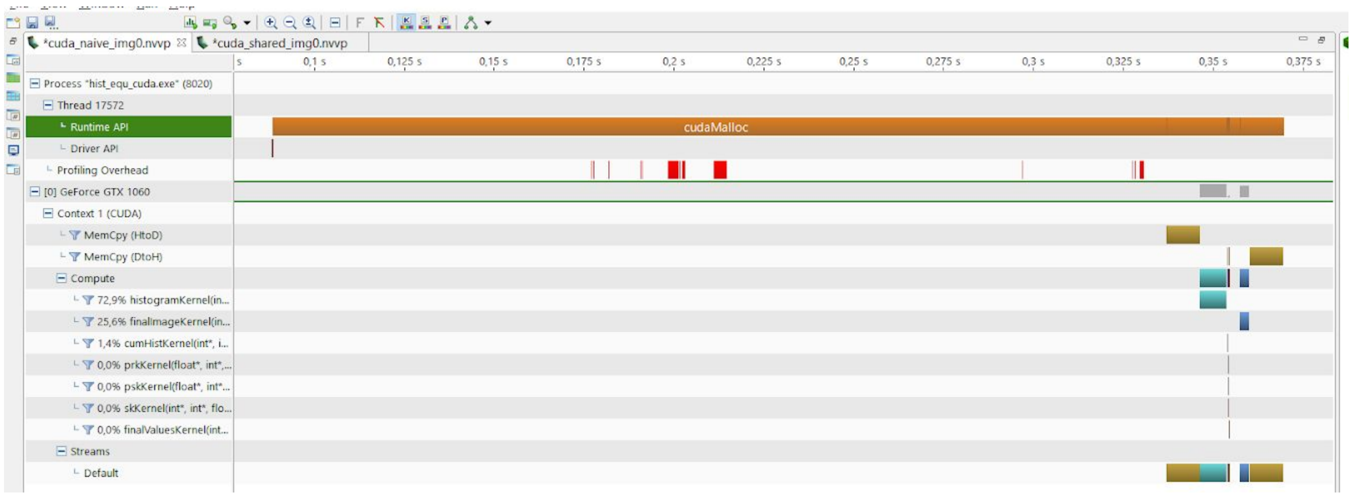
For a better exploration of the execution times, the Visual Profiler is used for comparing the implementation for the input size 20.000 (a square image with 20.000x20.000 pixels):



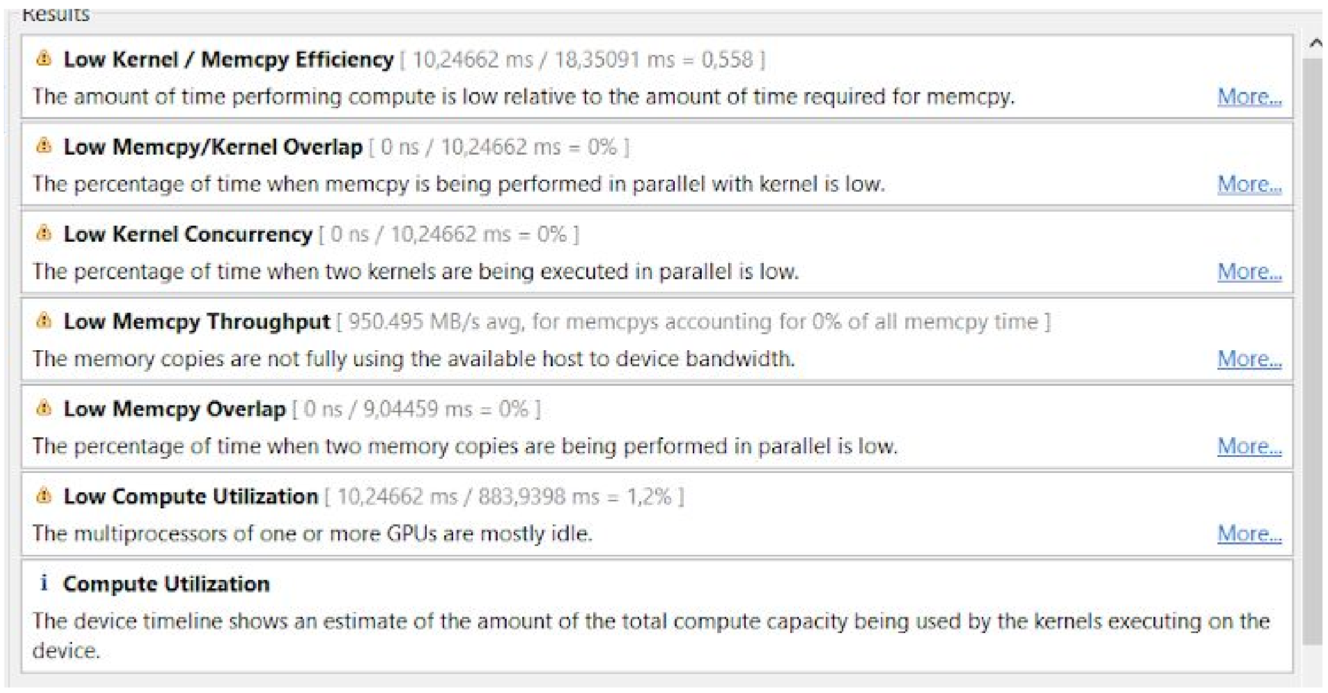
The visual profiler analysis results are shown below all implementations:

2.1. Naive version:

Execution times for all the steps:



The CUDA application analysis reports some problems with this implementation:



Results

Low Kernel / Memcpy Efficiency [10,24662 ms / 18,35091 ms = 0,558]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Kernel Overlap [0 ns / 10,24662 ms = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low. More...

Low Kernel Concurrency [0 ns / 10,24662 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [950.495 MB/s avg, for memcpys accounting for 0% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 9,04459 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

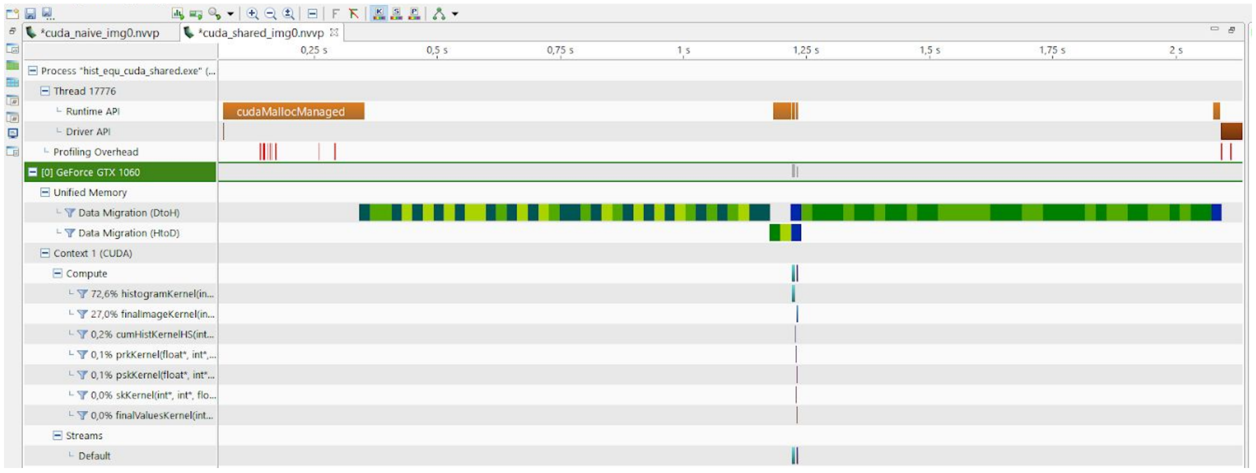
Low Compute Utilization [10,24662 ms / 883,9398 ms = 1,2%]
The multiprocessors of one or more GPUs are mostly idle. More...

Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

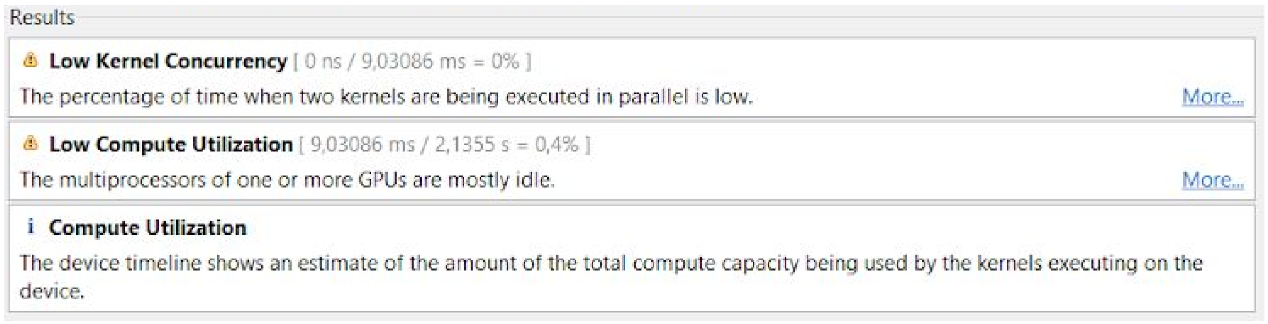
The naive implementation has several issues: inefficient memory copy to kernel overlap and vice-versa, low kernel concurrency, low memory throughput, low memcpy overlap and low compute utilization.

2.2. Shared memory version:

Execution times for all the steps and CUDA applicatio



n analysis are displayed below:



Results

Low Kernel Concurrency [0 ns / 9,03086 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [9,03086 ms / 2,1355 s = 0,4%]

The multiprocessors of one or more GPUs are mostly idle. More...

Compute Utilization

The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

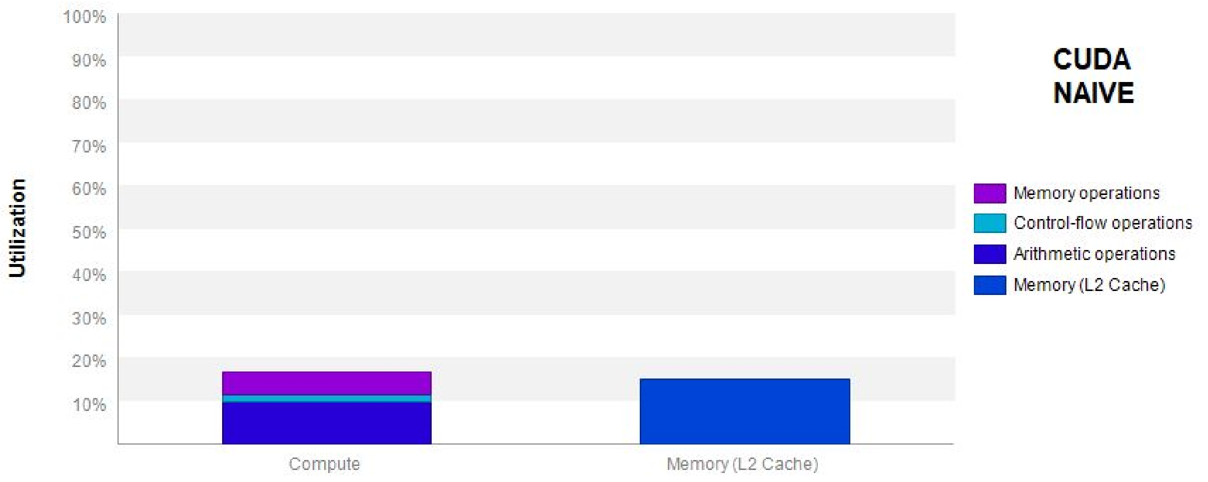
The shared memory version has only two problems, as taken from the visual profiler: low kernel concurrency, because some kernels only have one block and 256 threads and low compute utilization because most of the cores of the GPU are not used.

Comparison between naive and shared memory versions of histogram kernel:

The first image below is the naive Cuda implementation of the histogram kernel and the second one is the shared memory version of the histogram. There is an improvement in occupancy for the shared memory version from 91.2% to 94.8%. There is also an increase in the local memory overhead from 85.7% to 92.3%.

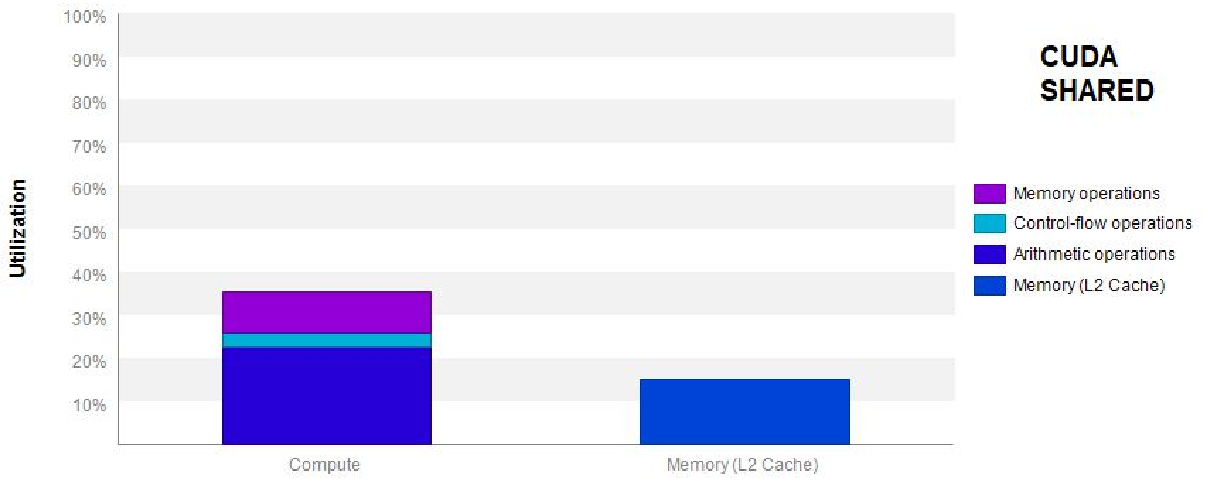
i **Kernel Performance Is Bound By Instruction And Memory Latency**

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



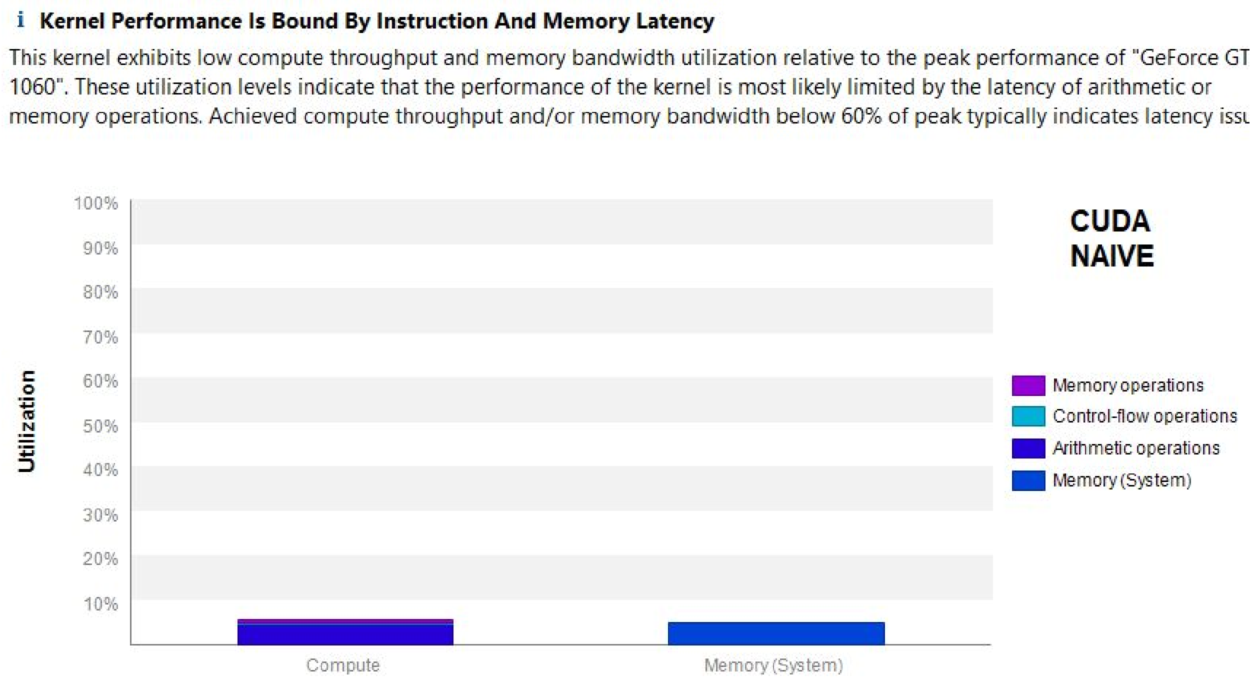
i **Kernel Performance Is Bound By Instruction And Memory Latency**

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

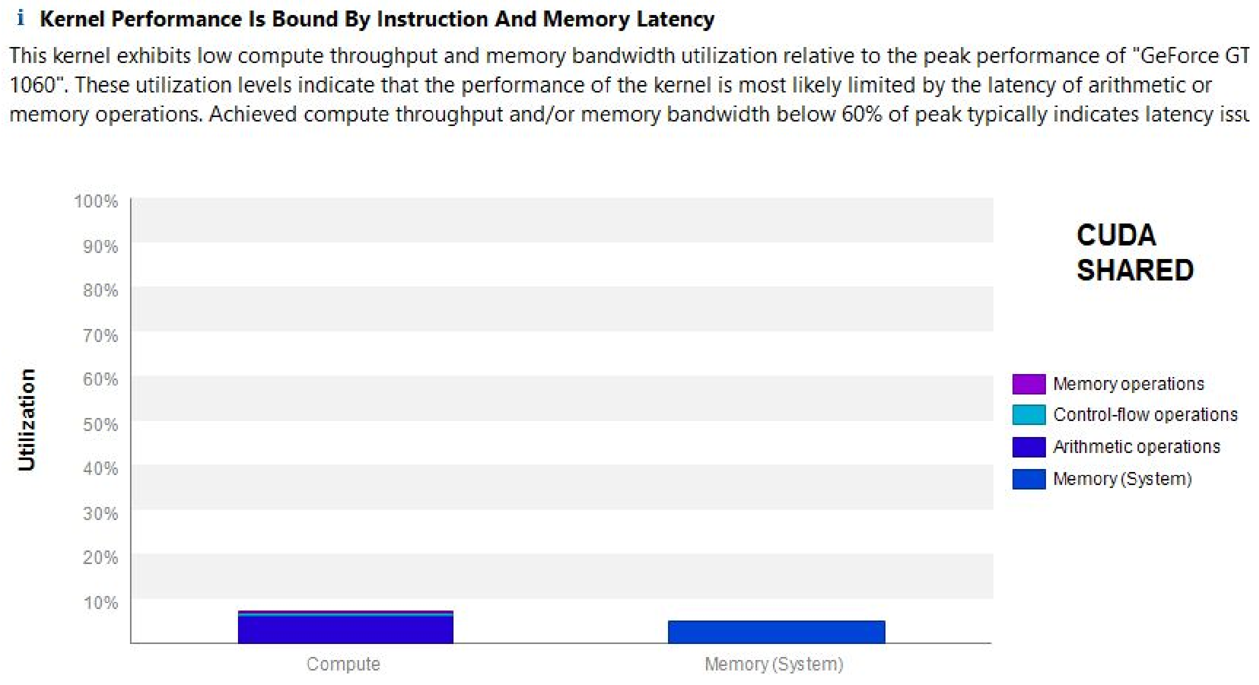


Comparison between naive and shared memory versions of the cumulative histogram kernel:

The first image below is the naive Cuda implementation of the cumulative histogram kernel and the second one is the shared memory version of the cumulative histogram. There is an improvement in occupancy for the shared memory version from 7.1% to 12.4%. There is also an improvement in execution time.


i **Kernel Performance Is Bound By Instruction And Memory Latency**

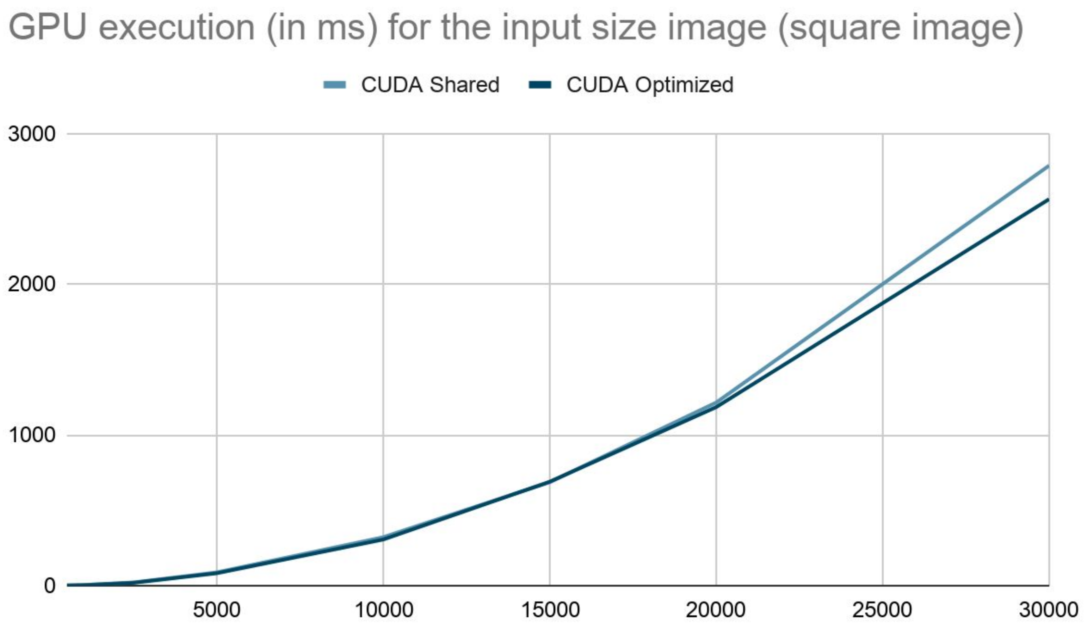
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GT 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issu


i **Kernel Performance Is Bound By Instruction And Memory Latency**

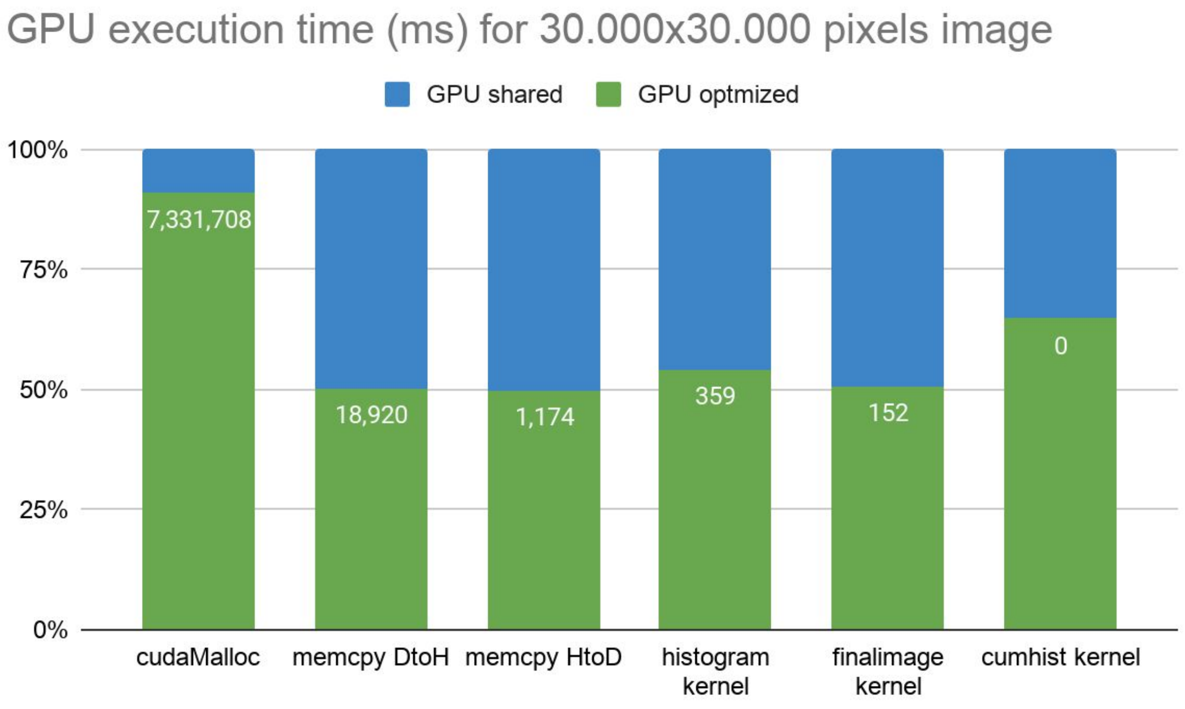
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GT 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issu

2.3. Third experiment - optimizations for GPU implementation

The execution time comparison between the optimized and shared memory CUDA implementations is shown below. The implementations offer similar execution times:



To explore better, the Visual Profiler is used for comparing the implementation for the input size 30.000 (a square image with 30.000x30.000 pixels):



The chart shows little difference between the two implementations for the input image.

The histogram implementation has changed in the optimized version and there is a slight increase in occupancy from 94.8% to 94.9%. The execution time is a little slower.

The cumulative histogram implementation has changed in the optimized version and there is a slight increase in occupancy from 12.4% to 12.5%. The execution time is slower, but the difference is negligible.

The kernels that were executed in parallel had a very low execution time and the difference between the shared and optimized versions are also negligible.

Conclusion

The resources of a system to solve a problem are limited. Parallelization can help to improve the execution time of some developed algorithms. The CPU offers high computing power but is limited in the number of cores (usually from 2 to 16). The GPU advantage is that it has a higher number of processing units (from 1000 to 10.000), but has a lower computation power. For image processing tasks, the GPU has advantages over the CPU implementation.

In the case of this project, the histogram equalization, the parallel GPU offers, as expected, better execution times than the CPU implementation, making it a better option. The GPU used is from Nvidia and uses CUDA. The naive implementation shows that the algorithm of histogram equalization can be parallelized with better results. Using the unified memory, a disadvantage of CUDA processing is the memory transfer from the host to the device and vice-versa. To improve this, global and shared memory is used. The unified memory, used incorrectly, gave as unexpected results, and the global and shared memory version had worse execution time. The GPU naive offers better execution time when the input image has an image width and height over 5000 pixels. The shared memory version has optimized memory usage and occupancy, but comes with lower execution time.

Further optimizations have been made. The algorithm for histogram uses a per-block computation and it is combined at the end, the cumulative histogram is a scan algorithm and uses an improved three version reduction. Some kernels are executed in parallel because they don't use or alter the same data. These show a small difference in optimization and the execution time is slightly improved for images with width/height > 20000 pixels. We expected better results, but the small difference still counts as improvement.

In conclusion, the parallel GPU implementation shows a better execution time for a bigger input image size. It can be used on applications where time is an important factor, regarding memory necessities.

References

[1] Open Source Computer Vision, found at
https://docs.opencv.org/master/d1/db7/tutorial_py_histogram_begins.html

[2] A Tutorial to Histogram Equalization can be found at
https://medium.com/@kyawsawhtoon/a-tutorial-to-histogram-equalization-497600f270e2

[3] Further Optimization in histogram CUDA code | Fast implementation of histogram in CUDA, found at cuda-programming.blogspot.com;

[4] Parallel Prefix Sum on the GPU (Scan), by Adam O'Donovan, found at umd.edu.