Parallel Computing 2023-2024
Final assignment

Matteo Trippodo
matteo.trippodo@edu.unifi.it

Abstract

This assignment has the aim to apply CUDA's potentialities for a task of kernel image processing. Since GPU are developed to fasten operation like matrix multiplications, in application as the considered one, the use of CUDA based code allows for severely high values of speedup, with respect to a simple sequential implementation.
Throughout this assignment the speedup performance are evaluated in multiple settings: considering images of different resolutions, using various block sizes and also assessing the impact of the time needed to copy back and forth data from host to device memory.

1. Introduction

Kernel image processing is the task of applying a given filter to an image, resulting in a output image in which some characteristics are enhanced or processed in specific ways. The filters can have various sizes, even though they are usually small sized (like 3 or 5 on a side) and have different definition based on their main purpose; for instance, we can have sharpen filters, edge detection filter or even Gaussian blur filters.
A filter can be seen as a simple matrix and usually it assumes very small sizes compared to the images is going to be applied. In our case are considered 3x3 and 5x5 filters.
Applying a kernel (filter) to an image practically means to compute the convolution between the pixels of the image and the matrix that represents the kernel.
The convolution operation can be defined as:

$$g(x,y) = \omega * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j)$$

where, in our case, f represent the image to process and ω is kernel filter we apply to it.
The basic idea behind the convolution is that this operation allows to find matching patterns between two signals and based on this general definition, the convolution operation can have lots of applications, mainly in the information and signal's fields.
In the specific case of kernel image processing convolution can be seen as we move the filter matrix above the image, multiplying the value of pixels and kernel's elements and then aggregating the result, using a sum. To reach the final result, this is process is repeated until all the pixel of the images have been processed.

2. Implementation

The implementation starts around the definition of the convolution operation and the items and tools we need to compute it.
We firstly have to define the kernels filter matrices. In the code the choice of the filter to be used can be done by the user modifying a constant in the initial part of the code.
The list of filter matrices and the relative associated numerical values can be read in the appendix 7.1.

As stated earlier, convolution is implemented by adding each element of the image to its local neighbors, weighted by the values of the kernel filters. This is derived directly by the mathematical definition of discrete convolution, recalled in the introduction, that, actually, it's not too far from a matrix multiplication. In fact, based on how we defined the operation it doesn't seem a proper matrix multiplication, but, actually, we can always define a proper matrix to convert the convolution operation into a matrix multiplication, and this intuition is often used in implementations for fields like Deep Learning.
Since this is not the aim of the current assignment, the convolution has been implemented following the previous definition, but this gave us good insight to expect great improvements when using parallelization for a case of kernel image processing.

Another important thing, when dealing with practical implementation of convolution is to take care of the corner handling. In fact, many times the kernel needs to use pixels that actually reside outside of the image's edges, and therefore in those cases we need to use some ghost cells.
In the considered implementation, the pixels out of the edge are actually not used to compute the convolution, so that,

only the pixel that actually belongs to the images are considered, while the other contributes are neglected, as if we would have pixels with neutral values (in this case zero).

2.1. Sequential code

The function `KernelProcessingSequential` takes as inputs the input and output image pointers, the sizes of the input image, beyond to the filter's the pointer.
The sequential implementation is done using three nested loops to consider every pixel across length, width and channel dimensions. For each output pixel we define a counter value, initialized to zero. Then, we compute the multiplication of a pixel and its neighbor pixels with the kernel filter. Then we sum the various contributes and we modify the value of the respective output pixel.

2.2. Parallel code

The parallel implementation is made up of three different functions or CUDA kernels: one that only utilize the global memory, one version that, instead, exploits tiling to store part of the image inside the shared memory, therefore reducing the number of readings from global memory, and a last version, that add to the previous approach also the use of the constant memory to store the kernel filter.

`KernelProcessingGlobal` has a similar implementation to the sequential version and extends it mapping the threads of the grid to the pixels of the output image. This is done using:

```
int col = blockIdx.x * blockDim.x +
    threadIdx.x;
int row = blockIdx.y * blockDim.y +
    threadIdx.y;
int channel = blockIdx.z * blockDim.z +
    threadIdx.z;
```

In this way each thread reads from the global memory a subsections of neighbor pixels of the input image, multiplies them for the kernel matrix and updates its own output pixel.

Note that, denoting with w the size of the filter, for each pixel, in this version, we do w^2 access to the global memory to read all the values of the filter and we perform two floating point operations (the multiplication between pixel and filter, and the sum to the partial value of the output).
Therefore, if we mainly consider the cost of accessing the global memory, the cost to process the entire image can be represented as $\mathcal{O}(HW(2w^2+1))$, where H, W identify the height and width of the image respectively.

To reduce the cost of the processing and, therefore, improving the performance of the code the best choice is to exploit the shared memory of the GPU. Every thread of the same block sees the same shared local memory, which is accessed in a negligible time if compared with the time need to reach the global memory.
This is the reason to extent the naive approach to convolution using tiling. In the kernel `KernelProcessingTiled` the image is accessed and processed in tiles, where the pixels of the tile are stored in the shared memory reducing the time needed to produce the output.

In the followed approach, the loading of the tile pixels on the shared memory is made mapping the thread on the output. This means that the dimension of the tile is bigger than the dimension of the block of threads. This implies that for each thread to have enough data to produce the output some of them need to load more than one value into the shared memory. So, inside the CUDA kernel we need to implement two separated loading steps.
In the first step each thread loads a pixel and stores it inside the tile data structure, in the second step we consider an offset of TILE_WIDTH * TILE_WIDTH and we replicate the process, but this time only the first threads of the blocks do a second upload, resulting in an unavoidable divergence of threads execution inside a given block. As happened before, we use some ghost cells inside the tile to account for the boundary handling. In fact, if the tile have to include some pixels that are actually outside to the image's edge we just load a neutral value.
Moreover, it's important to highlight that after each of the two steps we need to be sure that every thread have finished its load and therefor we need for a synchronization mechanism, using `synchthreads__()`.
After the loading phases, the convolution is performed as in the previous cases, but this time using the tile information that are stored in the shared memory.
Defining the width of the tile as $w_T = w_b + w - 1$, that for the sake of simplicity is just considered to be the a square, the total number of accesses to the global memory is scaled down by a factor w_b both for width and height. Therefore, the total cost for processing an image using tiling can be represented like $\mathcal{O}(\lceil \frac{H}{w_b} \rceil \lceil \frac{W}{w_b} \rceil (w_T^2 + w_b^2 w^2 + w_b^2))$.

The last improvement to speed up the code execution is to avoid completely to access the global memory when computing the partial results. This can be done exploiting the constant memory inside the GPU. The constant memory is a part of the GPU memory that is accessible by all the threads of the grid and allows for read-only operations. Since once the filter is chosen it's fixed, it can be seen as a constant of our code and therefore stored in the constant memory, before to call the kernel that executes the processing. In this ways the cost can be further reduced to something like $\mathcal{O}(\lceil \frac{H}{w_b} \rceil \lceil \frac{W}{w_b} \rceil (w_T^2 + w_b^2))$.

3. Hardware

For executing the code, I utilized a computer equipped with an NVIDIA GPU, specifically the NVIDIA GeForce RTX 4060 Ti, which belongs to the Ada Lovelace architecture family.
The main characteristic of this GPU are:

- Compute capability: 8.9;
- Total Global memory: 8 GB;
- Shared Memory per block: 48 KB;
- Registers per Block: 65536;
- Warp Size: 32;
- Maximum number of threads per block: 1024;
- Number of Streaming Multiprocessors (SMs): 34;
- Maximum number of threads per SM: 1536;
- Clock Rate: 2.565 GHz;
- Memory Clock Rate: 9001 MHz;
- L2 Cache Size: 32768 KB;
- Memory Bus Width: 128 bits

4. Profiling

In order to understand better how the code works and which are the instructions in the code that require more computation, or where the more time is spent, is always useful to do a proper profiling of the code.
In our case not only the execution times of the various kernel versions are gathered, but also, the impact of loading the data from and to the device is assessed. This is very important in such implementation since, as we discovered, most of the time is spent transferring data rather than executing the kernel, since it's very lightweight and it can be speedily executed by any decent GPU.
More precisely, what have been done is to use the NVIDIA Nsight System tools to have a visual understanding of the execution times of the code.
What has been noticed is that in the head of the execution there is a `cudaMemcpyToSymbol` instruction that usually requires a lot of time, which is used to copy the kernel filter in the constant memory. Then, the most of time is spend in the `cudaMemcpy` operations to copy the image from and to the device, while the actual execution of the kernel code represents a negligible part of the execution time.
In addiction, surprisingly, regarding the execution times of the three versions, on average the global version seems to be faster than the other one, with tiled+const that is slightly better the just tiled version. This phenomenon has been later studied more deeply using the Nsight Compute tools.

Studying the code execution using the profiling tool what came out is that a crucial part of the execution time of the kernel sometimes is the Queue Time. In fact, when we measure the execution time inside the code using instructions to record the instants of start and end of the kernel we cannot totally understand the actual underlying behavior of the code. The reason of that is because the time we record is actually made of three parts:

- API time: the execution time of the CUDA API call on the CPU used to launch the kernel;
- Queue time: the time between the launch call and the kernel execution;
- Kernel time: the kernel execution time on the GPU

The API time is usually bigger for the global version and smaller for the other two. But, from what has been observed, for the kernel time of the tiled+const version often suffers from bigger queue times; this means that there is a bigger latency from when the kernel is called to when is actually start its execution.

Thanks to the use of the profiling software an useful insight has been to use the pinned memory to store the data before transfer them to the GPU. This consideration allowed for less time needed to load the data into the GPU therefore reaching higher level of speedup.

Using Nsight System, keeping fixed one of the resolution (for istance 360p) and executing the three kernel versions, what results from it is that only the 12% of the execution time is actually due to the execution of the kernels, while around the 88% is about transferring data from host to device and vice versa. For the kernels part, around the 38% of the time is used for the tiled version, the 31% for the tiled+const version, while the kernel that uses the global memory occupies the remaining 30% of that time.

5. Experiments and results

Considering the explained implementation some experiments have been conducted varying the size of the kernel's blocks and the size of the filter used for the processing. In order to maximize the work of the threads, kernel with block size of 16x16 = 256 are considered (100% theoretical occupancy), but also blocks of and 32x32 = 1024 that allow for a 67% theoretical occupancy. .
The considered benchmark was made by pictures of 4 different resolutions and with the same height-width ratio (16:9). In fact, we tested the code processing 360p, 720p, 2K and 4K resolution images.

In the experiments, we kept fixed one of the resolutions at a time and we executed each parallel kernel 100 times, estimating then the average execution times, with and without considering the time need to load the data from and to the GPU device, and an index of variance across all the various iterations.
Actually, since has been noticed some not negligible variability inside the execution times of the various kernels, the final results are obtained considering some starting warm up executions and then recording the execution times of the last 100 executions.
Moreover, a caution that helped to identify more correctly the execution times was to introduce some synchronization point before and after the execution of the kernels, so that the each execution instance is less dependant from what happened before. Introducing this device sync point reduced a lot the latency to launch the kernels.

The results are summarized by the bar plots in the in the appendix (section 7.3).
The first thing we may notice, as also proven from the profiling tools, is that the tiled version doesn't perform as good as we may expect. In fact, when just the actual execution time is considered, the tiled version result to be slower then the global one, while the tiled+const leveraging the use of constant memory seems to perform as well as the other, if not even better.
The reason behind this behavior, after analyzing the code execution with Nsight Compute it should be the uncoalesced shared memory access. The tiled+const performs better since it exploits also the constant memory but still the profiling highlights its low level of memory throughput since the access to the pixels data is not efficient.
Another unexpected fact ease to detect is that, considering just the kernel execution times, the speedup level increases with the resolution, a part from the 4K images.
The reason of that is a low compute throughput, which means that the compute pipelines are under-utilized and it's probably due to the fact that not enough warps are issued to the scheduler. In fact, as the profiling tools highlights, there is an high level of warp stalls that reflects in a low SM occupancy. This should be mainly caused by long scoreboard stalls (i.e. something related to a L1 memory operation) and barrier stalls, related to the synchronization of the threads or even MIO (Memory Input/Output) stalls.
So, in general, the most of the problems seems to be due to uncoalesced storage of the data, that reflects to uncoalesced global access and uncoalesced shared access, and sometimes to a not negligible shared bank conflict. All of this brings the tiled version to perform worse than the others. This is exacerbated for the higher resolution image that also achieve high level of no-eligible warps due to various cycle of stall, resulting in lower speedup.

Analysing the plots we can also understand that most of the execution time is due to transfer of data between host and device, as already proven by the profiling tools. What we can see from the plots and the numerical values is that the time needed to execute any of the parallel kernels it's around one order of magnitude smaller than the sequential version, bringing to extremely high values of speedup, primarily if we consider the times deprived from the loads.
In our case, the values of speedup vary from around of 10 to 30 when considering the loads and from 10^2 to $4x10^2$ without them.
Moreover, when we consider the total kernel execution times, the three versions global, tiled and tiled + const usually doesn't differ too much for a given resolution. Overall, the total speedup time always increases with the size of the images, since the loading part of is the more dominant in the execution .

Finally, we can see that the trend of speedup both varying the size of the kernel and the size of the block doesn't change too much. A curious thing to notice though is that when the size of the filter increase from 3 to 5 the values of speedup are slightly increased, since less memory operations are performed and bigger work loads are issued to the scheduler.

6. Conclusions and final considerations

In conclusion, we have seen how some kernel-based processing code can be implemented exploiting the definition of convolution and how the code can have a speedup with the use of parallelization. The use of multiple threads in the GPU allows for high values of speedup that increase proportionally with the resolution of the images. The most of the execution time is occupied by the transfer of data though; this means that further development and improvement of this assignment could work on that aspect. In fact, CUDA allows, creating multiples streams, to execute different kernels while loading data from and to the device, bringing to even higher level of speedup. This could be done dividing the image in chucks and assigning each chunk to a different steams and letting the diverse stream to overlap while dealing with the operations of executing the kernel and when transferring data.
Moreover, it's worth to remember that to manipulate the images during the assignment the library OpenCV has been used and, specifically, to access the images the method imread() has been used, which saves the image in a Structure of Array manner. In fact, the pixel values are stored using triplets of R, G, B (actually, in the practice, as it's stated in the OpenCV documentation, the decoded images have the channels stored in B, G, R order, but this doesn't affect the implementation, since is managed

internally by the library). This way to store the image doesn't allow for a completed coalesced access to the pixel values, in fact neighbor threads cannot access to near data inside the memory, but they need to account for a certain stride, given how channels are memorized.
Since, as we explained, the main reason for the unexpected behaviours we assessed is due to uncoalsced access to global and shared memory, a future extension the project it could be also to pass to a more efficient AoS representation and storage of the pixel values for both gloabal and shared memory, allowing for a better coalescence and exploit the memory burst.

7. Appendix

To avoid to include verbose annotations or deeper details that are useful but not mandatory to appreciate the work done, those are included in this appendix.

7.1. Kernel filters

The kernel filter included in the code are:

1. Identity filter: it's a filter that doesn't alter the input image and it can be represented though the identity matrix.
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2. Sharpen filter: this type of filter allows to enhance object boundaries, so that we can recognizes more easily the various elements inside an image.
A sharper filter can be implemented in various ways, in the code this is done using the following matrix.
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

3. Edge detector (Ridge) filter: this filter allows to highlight the boundaries of the objects inside of the images. This kernel can be represent through a matrix like the following one.
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

4. Gaussian filter: blurs the image using an gaussian function approximated by a kernel matrix. An example of matrix that can be used is the the following one.
$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

5. Box blur filter:it blurs the image doing an average of the neighbors pixel values. For a matrix of size 3 it can represented as:
$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

6. Unsharp masking filter: this type of filter is usually used to make less blurry the input image, creating a mask using a negative version of the original image. For the case of a kernel of size 5 can be represented as follows:
$$\frac{1}{256}\begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -4 & -16 & -24 & -16 & -4 \\ -6 & -24 & 476 & -24 & -6 \\ -4 & -16 & -24 & -16 & -4 \\ -1 & -4 & -6 & -4 & -1 \end{bmatrix}$$

7.2. Numerical results

In this section are shown the timing values obtained during the experiment phase. Each table contains the values for each different resolution and each parallel kernel version. Moreover, the values in the tables are separated to show the differences between times with and without considering the CUDA loads (`cudaMemcpy`) and also varying the size of the blocks and kernel matrix.

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (tiled + const)
640 x 360	0.00561747	3.6884e-05	3.86983e-05	3.72377e-05
1280 x 720	0.0230736	7.08664e-05	7.83085e-05	6.05259e-05
2048 x 1152	0.0549197	0.000136733	0.000164632	0.000125152
3840 x 2160	0.199497	0.000886563	0.000955515	0.000933115

Table 1: Execution times with block size 16 and filter size 3

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00561747	0.000395821	0.000394398	0.000409344
1280 x 720	0.0230736	0.000952619	0.00121527	0.000976629
2048 x 1152	0.0549197	0.00185842	0.00202055	0.00183799
3840 x 2160	0.199497	0.00612098	0.00618599	0.00616203

Table 2: Execution times with block size 16 and filter size 3, considering CUDA loads

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00884468	5.25298e-05	5.73246e-05	4.29256e-05
1280 x 720	0.0359099	0.000538587	0.000288432	0.000114461
2048 x 1152	0.0914317	0.00030745	0.000320981	0.000205765
3840 x 2160	0.325184	0.00105694	0.00123593	0.00102651

Table 3: Execution times with block size 16 and filter size 5

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00884468	0.000407791	0.000419144	0.000399017
1280 x 720	0.0359099	0.00146674	0.00121527	0.000976629
2048 x 1152	0.0914317	0.00202656	0.00202055	0.00190248
3840 x 2160	0.325184	0.00629104	0.00650731	0.00627723

Table 4: Execution times with block size 16 and filter size 5, considering CUDA loads

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00564314	3.96044e-05	4.6817e-05	4.04916e-05
1280 x 720	0.0229673	8.3308e-05	0.000104028	8.90476e-05
2048 x 1152	0.0549303	0.000185646	0.000231741	0.000188426
3840 x 2160	0.199681	0.000882345	0.00130754	0.00115144

Table 5: Execution times with block size 32 and filter size 3

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00564314	0.000441993	0.000425869	0.000390128
1280 x 720	0.0229673	0.000991701	0.00101714	0.000977477
2048 x 1152	0.0549303	0.00196183	0.00193119	0.00186631
3840 x 2160	0.199681	0.00605957	0.00649156	0.00632775

Table 6: Execution times with block size 32 and filter size 3, considering CUDA loads

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00873367	5.38958e-05	5.78583e-05	4.61631e-05
1280 x 720	0.0358671	0.000287163	0.000167291	0.00011515
2048 x 1152	0.0914897	0.000351332	0.000383196	0.00026387
3840 x 2160	0.324684	0.00145226	0.00183812	0.001416

Table 7: Execution times with block size 32 and filter size 5

Image size	Seq. exec. time	Par. exec. time (global)	Par. exec. time (tiled)	Par. exec. time (constant filter)
640 x 360	0.00873367	0.000360077	0.000359681	0.000347506
1280 x 720	0.0358671	0.00178722	0.00100794	0.000956879
2048 x 1152	0.0914897	0.00203911	0.00206952	0.001944
3840 x 2160	0.324684	0.0066958	0.00706885	0.00665065

Table 8: Execution times with block size 32 and filter size 3, considering CUDA loads

7.3. Speedup bar plots

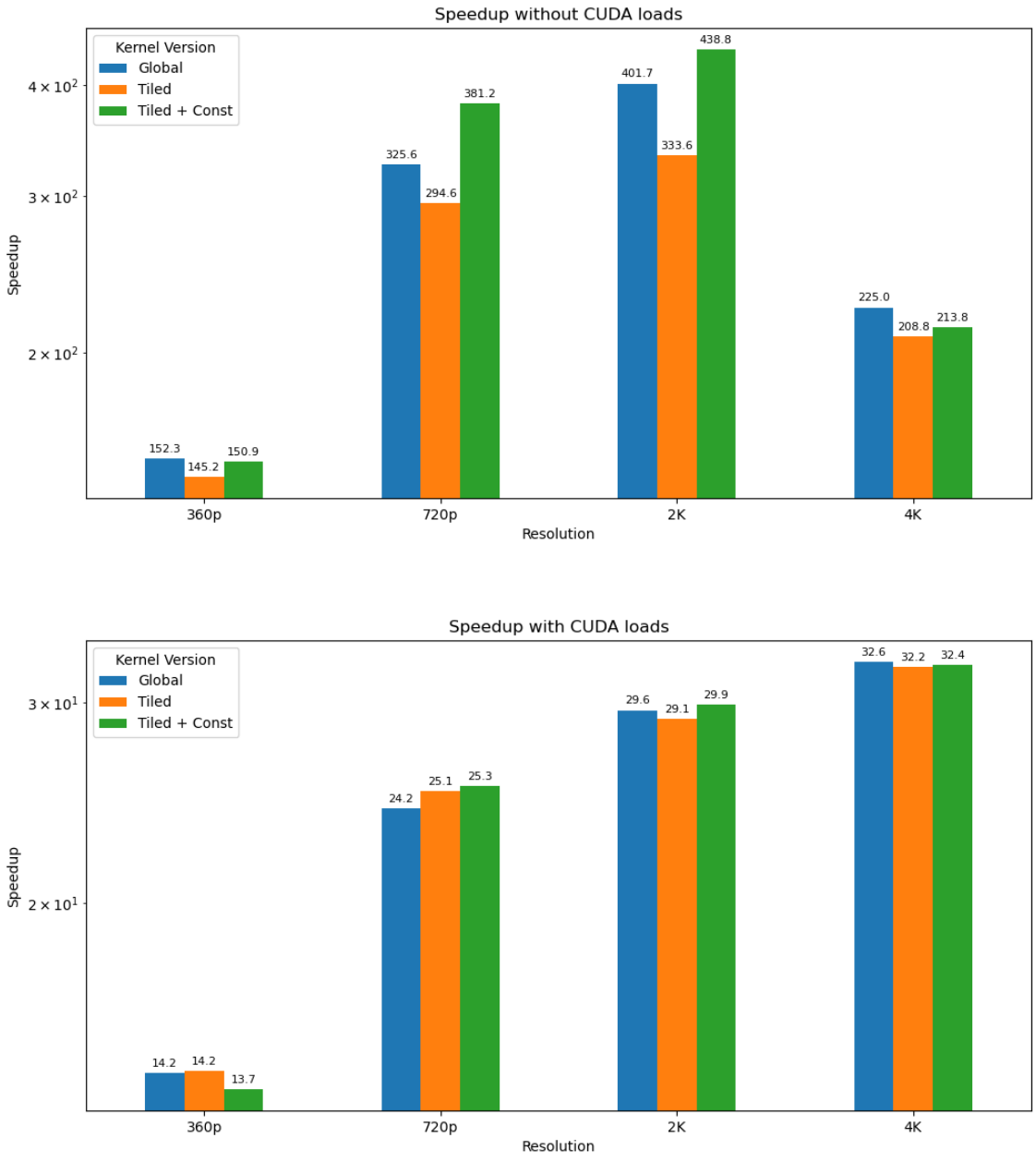

Figure 1: Speedup values for a (16,16) thread block and a filter matrix of size 3

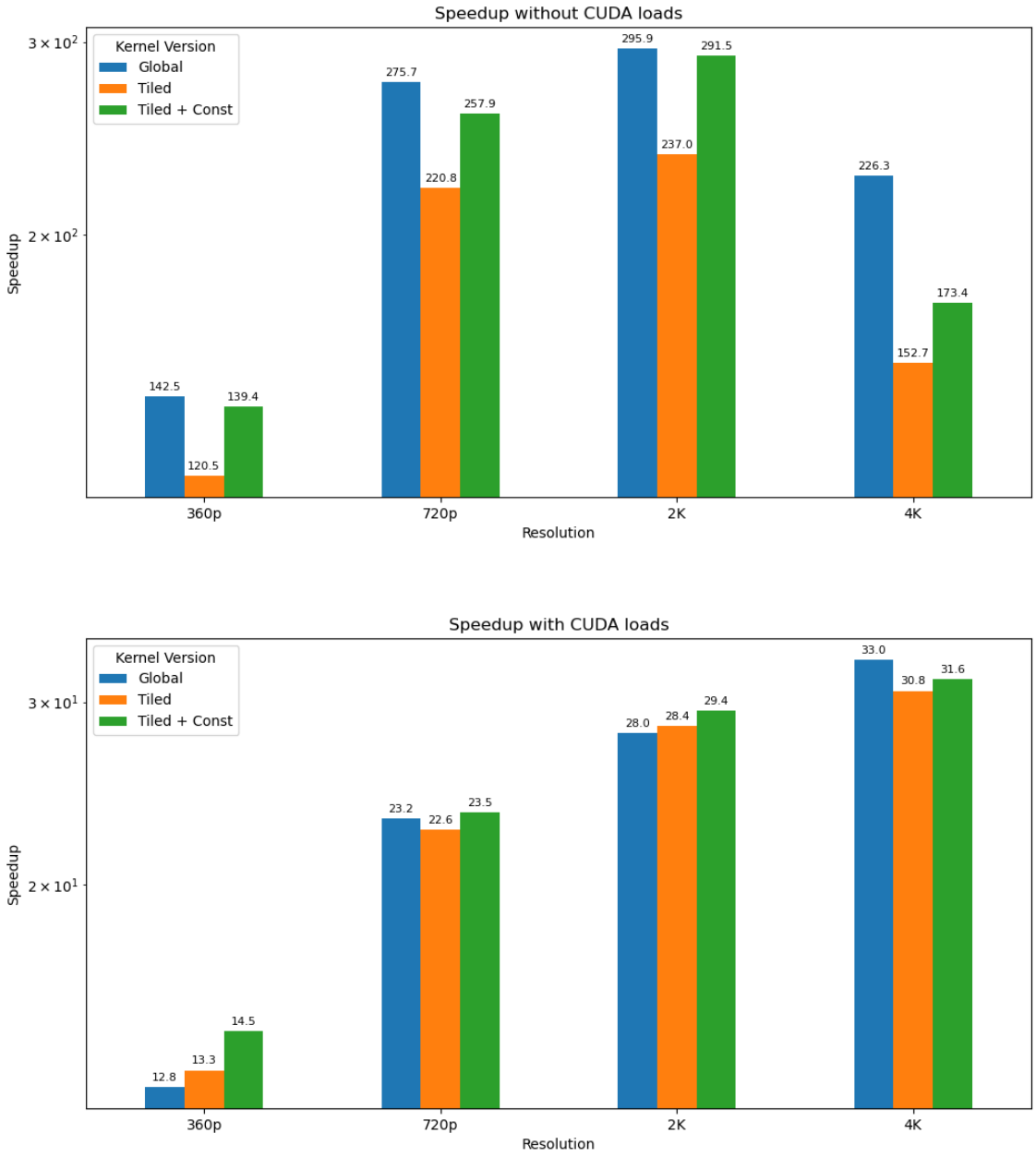

Figure 2: Speedup values for a (32,32) thread block and a filter matrix of size 3

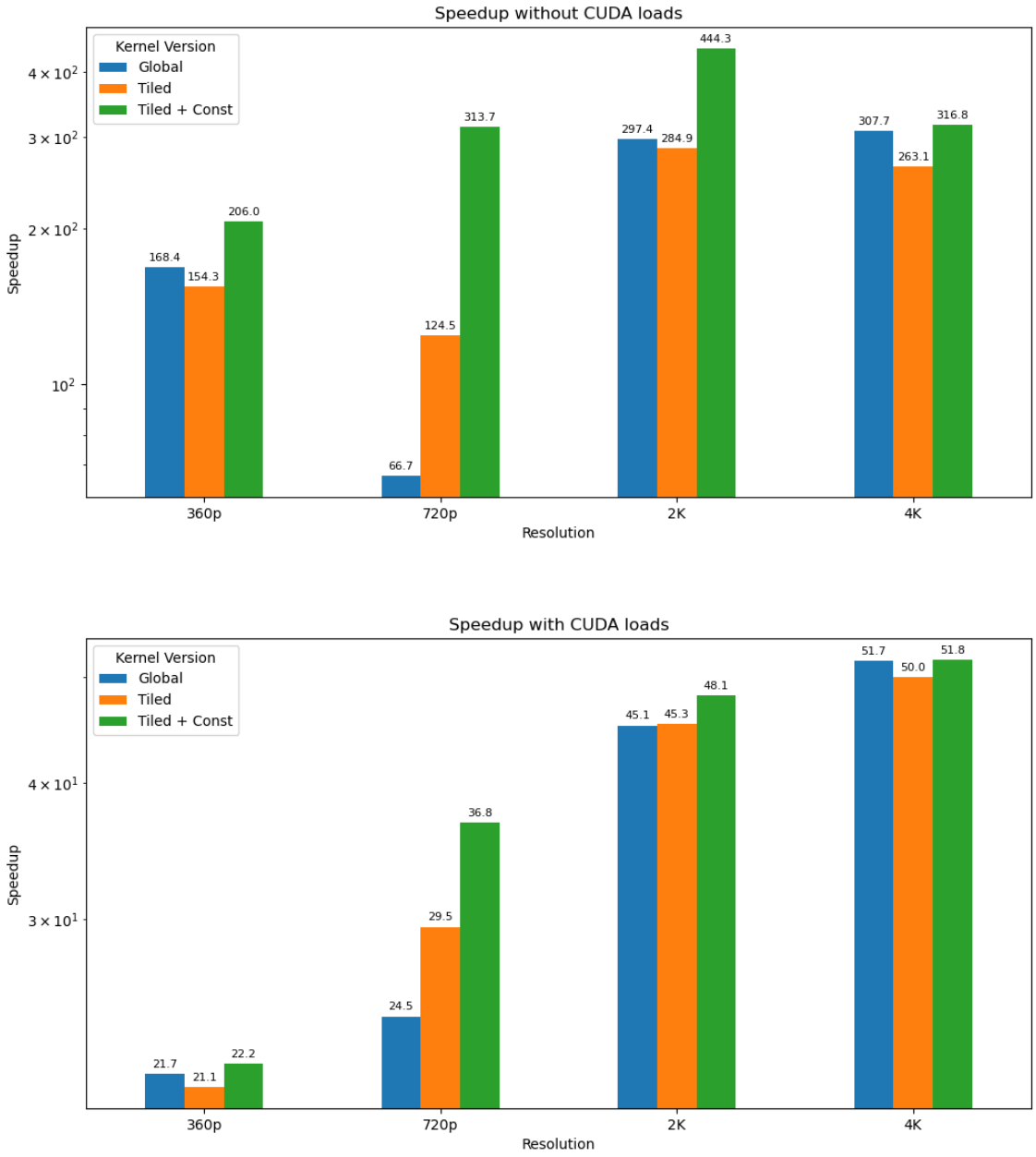

Figure 3: Speedup values for a (16,16) thread block and a filter matrix of size 5

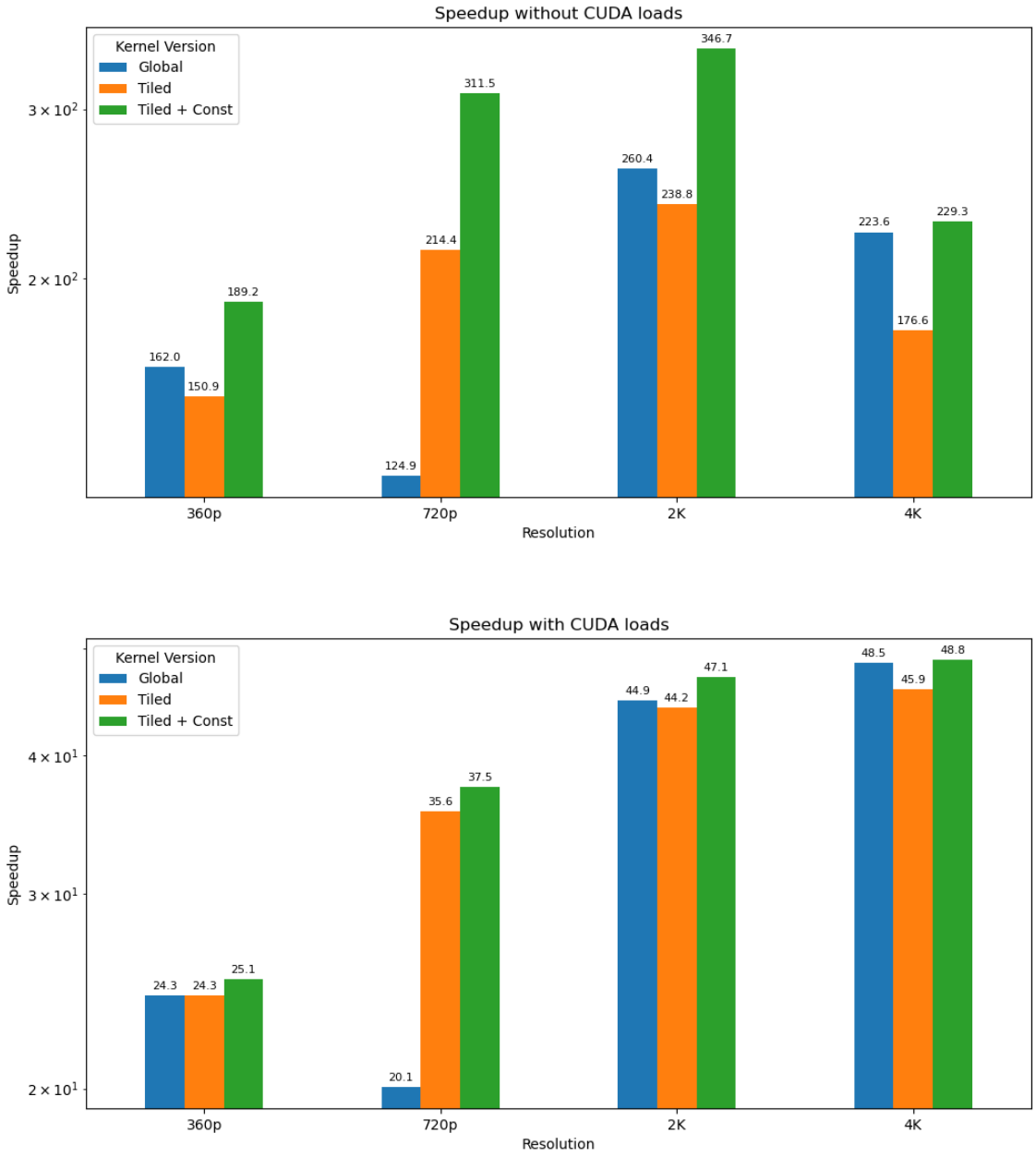

Figure 4: Speedup values for a (32,32) thread block and a filter matrix of size 5

7.4. Examples of processed images

Some example of how the images are processed using an Edge Detection filter.


Figure 5: 360p image before and after the processing


Figure 6: 720p image before and after the processing


Figure 7: 2K image before and after the processing


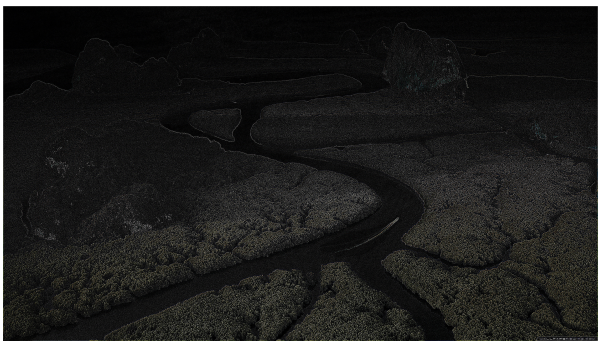

Figure 8: 4K image before and after the processing