Image Zooming and Upscaling Using CUDA

GPU Programming course
A.Y.2022/2023

Politecnico di Torino

Marchetti Laura [s294767]
di Gruttola Giardino Nicola [s298846]
Durando Luca [s303395]

Contents

Abstract

In this document an upscaling algorithm in CUDA has been implemented: it is a technique used to produce an enlarged picture from a given digital image while correcting the visual artifacts originated from the zooming process.

The zooming algorithm works on RGB images in PPM format: it provides as output a zoomed picture, cut-out from the original one passed through the command line, while simultaneously correcting its aliased behavior by implementing a convolution with a specific filter.

The user needs to set, through the command line, the RGB picture that has to be zoomed, the coordinates of the center of the selection zone and the side lengths of the selection mask. It is also left to the user to specify the filter to be applied in the convolution: it's both possible to load a custom kernel or create a Gaussian filter specifying its length and standard deviation.

1 Related works

1.1 The Pixel Replication Algorithm

The GPU version of the already existing algorithm named "Pixel replication" [1] (also known as the "Nearest neighbor interpolation") is developed in the proposed implementation of the zooming algorithm. As its name suggests, it replicates the neighboring pixels to increase them in order to enlarge the image: it creates new pixels from the already given ones replicating each pixel "n" times row wise and column wise. This Algorithm has the advantage of being a very simple technique to implement zooming but, on the other hand, as the zooming factor increases the resulting image gets worse.

1.2 Image Filtering Convolution Algorithm

For the purpose of improving the visual quality of the zoomed image it is implemented an image filtering algorithm based on the convolution operation [2], that can be applied to reduce the amount of unwanted noise. In order to implement the convolution between the zoomed image and a specified filter, the already known version of the "Image filtering through convolution" algorithm is adapted. In the original CPU implementation of the algorithm the convolution operation is performed by sliding the mask onto every pixel of the image using a double loop while simultaneously iterating over every element of the kernel mask for each pixel, making use of another double loop. The basic algorithm implements the convolution operation in a simple but unoptimized approach that can be improved, for example by exploiting the GPU parallelism concurrently with the tiling technique [3].

2 Proposed Method

In this section a detailed description of the proposed algorithm is given to the lecturer: some implemented procedures will be explained through pseudocode. The algorithm works in successive stages as described in the following:

- Kernel loading
- Image reading
- Output Image creation
- Image selection, enlargement and filling
- Image enhancement

The algorithm is divided into two parts: the first one focuses on the CPU, which is responsible for the reading of the image and the kernel loading; the second one is the GPU part, which is responsible for the image cut out, enlargement, filling, and enhancement. The inputs required by the application are based on the commands given by the user through the command line, and they are the following:

- The path of the image to be zoomed;
- The *-c* (*–custom*) flag, which refers to a custom kernel, followed by the file path of the kernel;
- The *-g* (*–gauss*) flag, which refers to a Gaussian kernel, followed by the length of the kernel and the Σ value, mutually exclusive with the previous;
- An optional v character, appended to the mode, which enables the verbose mode;
- An optional f character, which forces the use of global memory instead of the shared;
- The coordinates of the center of the cut-out area, in the form of x, y;
- The width and the height of the cut-out area;
- The zoom level of the image must be an integer bigger than 0, it represents the multiplier of the cut-out area dimension;

```
1    # Example of command line with gaussian kernel
2    # cutout center (100,100), zoom 2, dimensions 50x50, kernel size
   31, sigma 5
3    upsCu -g ./img.ppm 100 100 50 50 2 31 5
4    # Example of command line with custom kernel
5    # cutout center (100,100), zoom 2, dimensions 50x50, kernel file ./
   kernel.txt
6    upsCu -c ./img.ppm 100 100 50 50 2 ./kernel.txt
7    # Example of command line with global memory
8    # cutout center (100,100), zoom 2, dimensions 50x50, kernel size
   31, sigma 5
9    upsCu -gf ./img.ppm 100 100 50 50 2 31 5
10   # Example of command line with verbose mode
```

```
    # cutout center (100,100), zoom 2, dimensions 50x50, kernel size 31, sigma 5
    upsCu -gv ./img.ppm 100 100 50 50 2 31 5
```

2.1 Kernel Loading

The algorithm can choose between custom kernels loaded from files, or Gaussian ones, which are generated on the fly, taking as input the length of the kernel and the standard deviation.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

The kernel is generated by using the equation above, iterating over the matrix length, and then normalizing it. The normalization is done by dividing each value of the matrix by the sum of all the values of the matrix.

```
...
kernel[i][j]=exp(-((i - kernel_size / 2) * (i - kernel_size / 2)
    + (j - kernel_size / 2) * (j - kernel_size / 2)) /
    (2 * sigma * sigma)) / (2 * M_PI * sigma * sigma);
...
kernel[i][j] /= sum;
...
/* Load into constant memory */
cudaMemcpyToSymbol(d_kernel, kernel, dimKernel * dimKernel * sizeof(
    float));

```

Once generation is done, the kernel is loaded into GPU's constant memory, which allows the program to save time. This happens because constant memory is a special type of global memory with a peculiar cache that doesn't need to do as many coherency tests as the other caches. This is useful because every value of the mask is read at the same time by every thread of the program, and it is not modified during the execution of the program.
The *GaussLength* parameter must be an odd value from 3 to 127 sides included and the *GaussSigma* parameter must be a value from 0.5 onwards side included. The custom kernel must be a square matrix, and it has to be at maximum *MAX_KERNEL_DIM* long, which is set to 127 elements.

2.2 Image Cut-Out

The aim of this step is to select the part of the original image that has to be trimmed: the dimensions of the cut area are passed through command line and the script subsequently calculates the dimension of the output image.
The function carries out the logic explained down below:

```
...
img_out[tid] = img[starting_byte+row_offset+column_offset]
...

```

Where *tid* is the thread ID, *img_out* is the final image, *img* is the original image, *starting_byte* is the starting byte of the cut area, row_offset is the offset of the row of the pixel to be copied and $column\_offset$ is the offset of the column of the pixel to be copied. The implementation is basic, it is a simple copy of the pixels from the original image to the final one, done in parallel

using CUDA threads.

2.3 Image Enlargement and Filling

This step is the one that enlarges the image and fills the holes that are created by the zooming process. The process begins by creating as many threads as the bytes in the final image, and each one of them computes the value of the pixel to be copied from the original image. The holes are filled with the help of the pixel replication algorithm.

```
...
int stuffing = dimImgMid / dimImgIn * 3;
if (idx >= dimImgMid * dimImgMid * 3)
{
    return;
}
rowOffset = offset * dimImgOut * 3;
colOffset = offset * 3;
outputRowOffset = idx / 3 / dimImgMid * dimImgOut * 3;
outputColOffset = idx / 3 % dimImgMid * 3;
position = colOffset + rowOffset + outputRowOffset + outputColOffset +
   idx%3;
offsetScaledRow = idx / dimImgMid / stuffing * dimImgIn * 3;
offsetScaledCol = (idx / 3 % dimImgMid) / stuffing * 9;
scaled_img[position] = cutout[offsetScaledRow + offsetScaledCol + idx
   %3];
...

```

It replicates the neighboring pixels in order to increase them to enlarge the image, based on the zoom factor. It is the most basic technique to implement zooming, but it is the most efficient one since it does not require any computation.
In the first version of the algorithm the canvas gets filled with the enlarged image and black borders around it, which will be used to perform the convolution with the filter, since it needs a larger image than the one that is actually returned as output.

```
...
if ((row_i >= 0) && (row_i < inHeight) && (col_i >= 0) && (col_i <
   inWidth))
{
    in_img_shared[ty * blockDim.x + tx] = input[(row_i * inWidth +
   col_i) * 3 + color];
}
else
{
    in_img_shared[ty * blockDim.x + tx] = 0;
}
...

```

The final version of the algorithm instead gets directly the pixels needed and if the cut-out is not at the border, the program will use neighboring pixels to create a slightly bigger image for the convolution. If the cut-out is at the border, the nearest pixel will be used just for the border part. This final version produces an image which is really close to the original one.

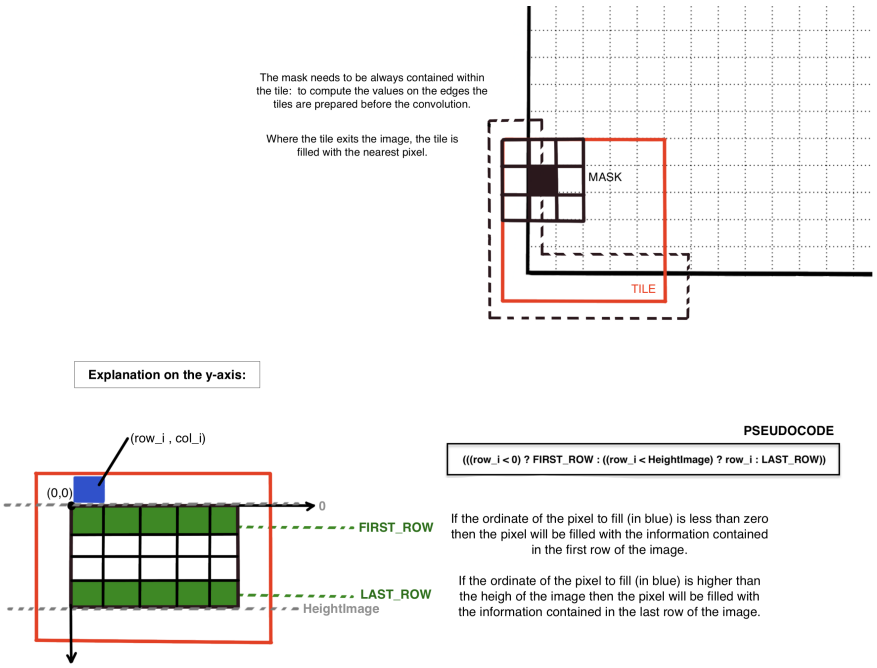

Figure 1: Border pixel replication (same happens for columns)

2.4 Image Enhancement

This step executes the convolution of the image with the kernel, in order to enhance the image and make it clearer.
Many versions of the algorithm have been tested, starting from only using global memory, to using shared memory and tiling technique.
The first version of the program, which uses only global memory, performs the operations slower than the other, but it is the most straightforward one used as workbench. The function remains available for the user, when needed. The second version of the algorithm, improves the performances of the algorithm, by exploiting the shared memory, which allows to have a faster access to the data. This happens because it is a cache located on chip, thus accessible by all threads of a block, providing a mechanism for threads to cooperate, while diminishing the access to global memory stored in the DRAM. These main problems arise when using shared memory: the fact that it is limited in size, it is not possible to use it for all the data needed and bank conflicts. The last issue is caused by the shared memory division into banks which can be accessed simultaneously by different threads only if they are accessing different registers. This means that if two threads are accessing the same bank, they will have to wait for the other one to finish, and this will cause a slowdown in the execution of the algorithm. The tiling technique is preferred because it uses the shared memory and the commodities and hacks that comes with it, avoiding conflicts. The image is divided into tiles, convolution is then performed on each of them and the result is the final image. This technique allows avoiding bank conflicts, because each tile is stored in a different bank, reducing memory traffic.
The basic version of the algorithm uses a kernel created by the *Gaussian_Kernel_Cpu* function stored in constant memory referenced by *d_kernel*. It computes the convolution using one thread for each byte of the image by dividing it in blocks of the same size. The convolution is performed per color channel, using blocks of pixels of the same dimension of the kernel, centered in the pixel to be processed. The result is achieved by multiplying the value of the pixel with the value of the corresponding element of the kernel, and then summing all the results.

The final version, with shared memory and tiling, decides if the image is large enough to be tiled by computing the width of the tile:

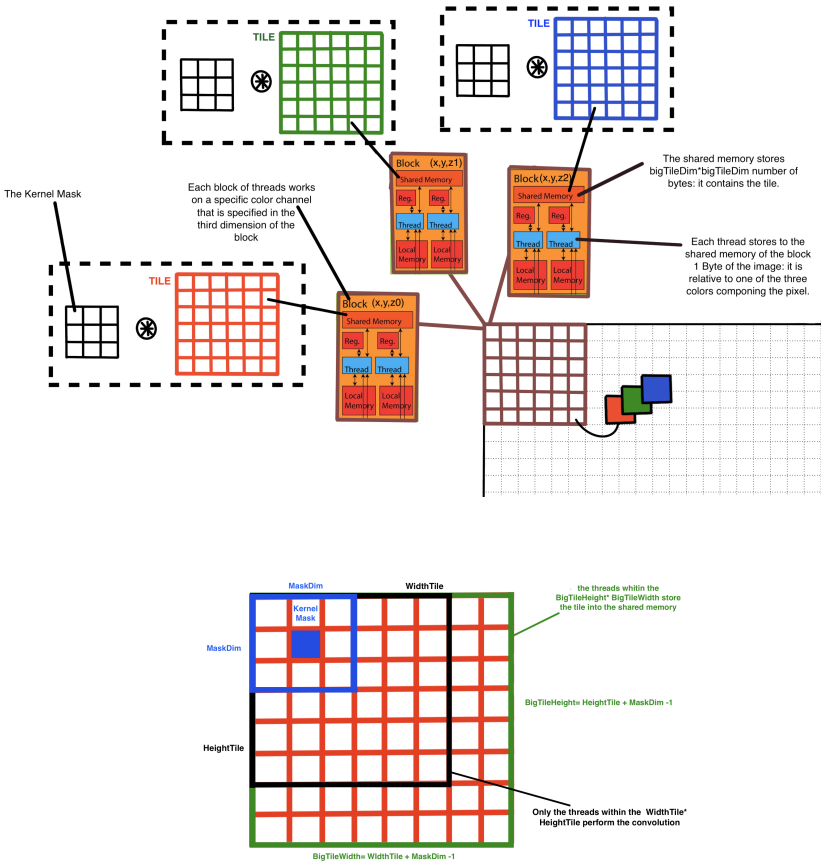

Figure 2: Schematic representation of enhancement process

```
...
heightTile = widthTile = \sqrt{maxThreadsPerBlock} - (maskDim - 1);
setTiling(widthImgOut, heightImgOut, &widthTile, &heightTile);
...

```

If the product of *heightTile* and *widthTile* is bigger than one, the tiling technique is performed, otherwise the global memory version is used because the tiling technique would not be efficient due to the overhead of the tiling process. The shared memory has to be filled first with the color information of the image, then used to perform the convolution with the mask. All the initialized threads of the block participate in filling the memory but afterwards only some of them participate in calculating the convolution with the mask for that block. The number of threads set for each block corresponds to the number of bytes allocated for the shared memory: in the project it is set to $bigTileDimX * bigTileDimY$.

2.5 Code implementation

Taken into account what has been said before, the actual implementation has a more direct way of performing the cut, enlargement and convolution. The initial version of the algorithm

did cut, enlargement and convolution in three different CUDA call, but this was not the most efficient way of doing it.
The final version, instead, performs the whole process in a single CUDA call, which is called by the main function, and it is the one that is actually executed and demonstrated. The program calls *globalCudaUpscaling* if the global memory version is used or *tilingCudaUpscaling* in case the shared memory version is possible.

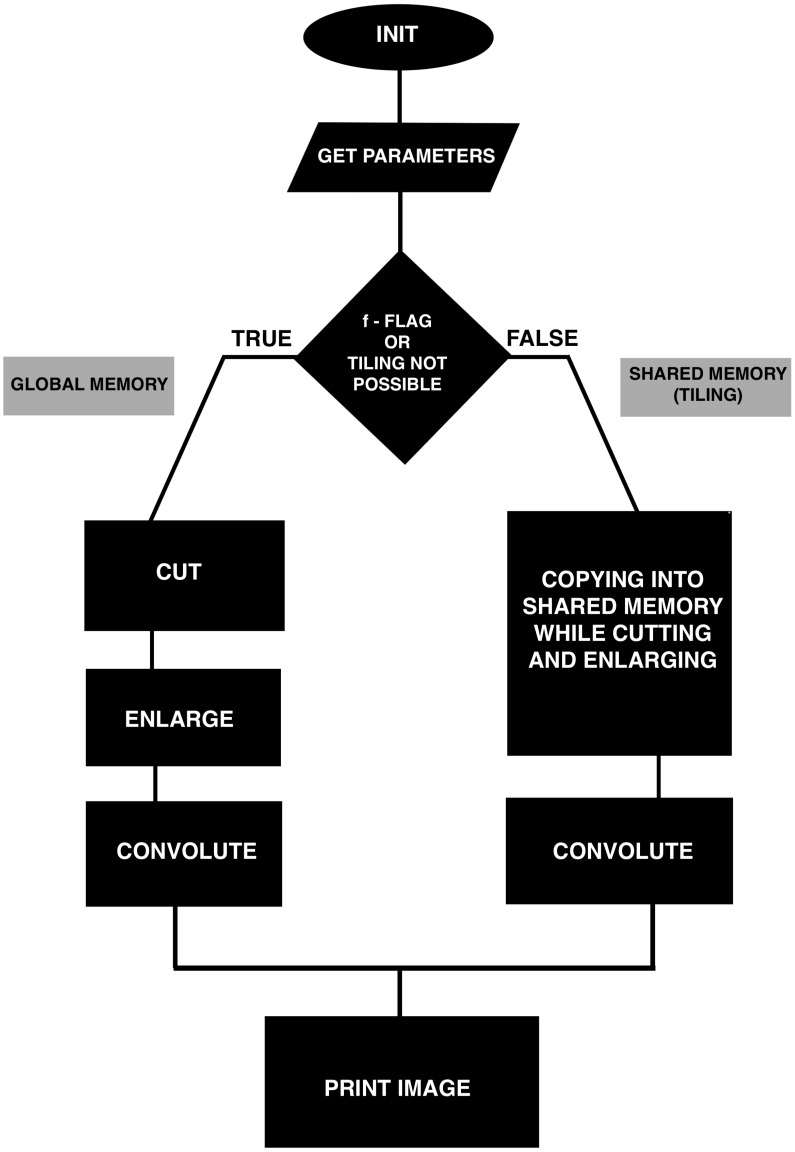

Figure 3: Flowchart

3 Experimental Results

3.1 Experimental setup

The project has been carried out using a Jetson Nano [4], a small single-board computer with a quad-core ARM Cortex-A57 CPU and a Maxwell GPU [4]. The Jetson Nano is equipped with 4GB of RAM and an SDCard for the Operating System and data storage. The operating system used is Ubuntu 18.04.

Figure 4: Jetson Nano

The project has been developed using the C++ programming language and the CUDA library. The CUDA library has been used to exploit the GPU capabilities and to perform the image processing operations in parallel.

3.2 Program's history

The project has been developed gradually. The first version of the project ($v0.5$) has been developed on January 19th. This version implemented the image processing operations using three different CUDA functions, one for each operation. This operating mechanism slows down the execution of the program because the image is copied back and forth three times between GPU and CPU memory, other than suffering from the overhead of the CUDA function calls and kernel creation. Moreover, there are black borders around the image, which create a worse looking image when the borders are enlarged. In $v0.5$ the final image has square dimensions, since the complete zooming feature has not been implemented yet.

The second version of the project ($v0.9$) has been developed on January 26th. It encapsulates the three image processing operations in a single CUDA function, which is called only once, making the execution faster and reducing the overhead of the CUDA function calls and kernel creation. This version of the project still implements the black border around the image. Moreover, it implements a complete zooming feature, which allows the user to zoom in by selecting a rectangular area of the image.

The third and final release of the project ($v1$) has been developed on February 10th. This version removes black edges around the image using the technique explained in the previous section, producing a better looking output when the image is zoomed near the edges.

3.3 Results comparison

All three versions have been analyzed and compared in terms of execution time and image quality. Moreover, the profiling of the code has been performed to understand the bottlenecks, while also trying to make fully use of the calculator's capabilities. These results are presented in the following table 1. The main notable difference in the table between the 3 version is the execution time of the program. This is due to the fact that the first version of the project uses

Version	Global/Shared memory	Time to execute (s)	Shared ld/st conflicts
v0.5 (19 Jan)	Shared	3.09	0
v0.9 (26 Jan)	Global	0.495	/
v0.9 (26 Jan)	Shared	0.460	0
v1 (10 Feb)	Global	0.148	/
v1 (10 Feb)	Shared	0.137	0

Table 1: Experimental results

three different CUDA functions, as explained in the previous section. This is also due to a poorer use of GPU capabilities. The second version achieves a speedup of 6.2x with respect to the first one, while the third version achieves a speedup of 21x. This speedup is achieved thanks to the substitution of the if-else clauses with the use of mathematical operations for computing of indexes.

The following [5] is the original image used to test the project. The image is a 640x426 pixels image, which is a typical resolution for a mid-range smartphone camera.

Figure 5: Original image

The following [6] shows the zoomed image using the last version of the project. On the left side, the original image is shown, with a zoom performed one the selected area, without the use of a filter. On the right side, the zoomed image is shown, with the use of a Gaussian filter of size $15x15$ and a standard deviation of 5.

Using the Jetson Nano to achieve maximum performances and to fully exploit the GPU capabilities, the Watchdog timer was disabled, which is a feature that automatically stops hanging processes. This feature is enabled by default, and stops the execution after 5s of activity of a single thread. The discovered limits of the Jetson Nano have been found to be a maximum resolution of 5000x5000, using a Gaussian kernel of size $15x15$, while keeping the same size of the original image ($640x420$), the maximum Gaussian kernel size which can be reached is found to be 111x111. Obviously these numbers are not the upper limit in every occasion, but they're the maximum possible for the image used in the test.

3.4 Using custom kernel masks

Gaussian kernel are used to blur the image, mainly to reduce the noise, obtaining a smoother picture from the convolution, ideal after enlarging it. Other types of more suitable masks can be used for other purposes. The examples shown down below are the results of kernels convoluted to execute embossing [5], edge detection, ridge detection [6] and darkening/lightening of the image.

Figure 6: Zoom in the original image & and zoomed image using v1

In figure 7 the embossed image is shown. The embossing is a technique used to create a 3D effect on a 2D image, by using a specific kernel. The 3D effect of the image shows the depth of the pixel. Every colour has a different depth, going from black, the deepest, to white, the narrowest.

Figure 7: Embossed image

In figure 8 the darkened and lightened images are shown. The darkened image is obtained by using a $3x3$ kernel which contains as center a value between 0 and 1, because it reduces the value of all the pixels in the image. The lightened image is obtained by using a $3x3$ kernel which contains as center a value greater than 1, because it increases the value of all the pixels in the image, given the multiplication used by the convolution.

The last two alternative kernels used in this project are for edge and ridge detection. They do the same thing, but the difference is that edge detection is more sensitive to changes, meaning that a sudden change in the colour of the pixel, remaining in the same color space, is detected as an edge. Ridge detection is less sensitive, it is more focused on changes between different colours, achieving a better result in detecting the edges of objects, or, in the case of the image used in the test, the edges between the sky and the trees, as well as those between the lights in the grass and the grass itself. The previous, instead, detects edges between each curve in the grass and the light.

Figure 8: Darkened image & Lightened image

Figure 9: Edge detection & Ridge detection

4 Conclusions

In this paper a technique for both zooming and enhancing a digital picture by utilizing GPU parallelism is proposed.

When the first CPU version of the algorithm was implemented parallelization was not achieved in the workload. A first non-optimized implementation on the GPU was then used: at the end of the procedure the program was enhanced by working on its bottlenecks like memory reading for the data transfer.

Concerning the employment of the shared memory for the tiling process, a complete absence of memory access conflicts was observed: GPUs with a computer capability greater than 2.0 are able to avoid conflicts in this scenario. This is due to the fact that shared Memory sends data in broadcast to all the threads accessing the same memory location.

In a future perspective, within the overall Upscaling algorithm implemented, the section concerning the zoom algorithm can certainly be optimized: a basic algorithm that simply replicates the adjacent pixels was utilized but, for example another type of algorithm could be used, able to interpolate the color values instead of replicating the already existing ones. If the CUDA Runtime API gets updated to support vectors in global memory of size greater than *MAX32_INT* then the algorithm can be easily extended to work on bigger images.

References

[1] Zooming methods. https://www.tutorialspoint.com/dip/Zooming_Methods.htm.

[2] L. Sterpone, C. De Sio, and J. Rodriguez Condia, "Convolution memory management," 2022/2023, slides for the lecture.

[3] ——, "Tiled convolution," 2022/2023, slides for the lecture.

[4] Jetson nano developer kit. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[5] Custom kernels (darken/lighten/emboss). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#custom-kernels.

[6] Custom kernels (boxblur/edgedet/identity/ridge/sharpen/unsharp). https://en.wikipedia.org/wiki/Kernel_(image_processing).