# 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 we implement an upscaling algorithm in CUDA: It is a technique used to produce an enlarged picture from a given digital image while correcting the visual artifacts originated in the zooming process.

Our 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 length of the selection mask which has a square odd shape. It is also left to the user to specify the filter to be applied in the convolution: it's both possible to set out a custom kernel or to create a Gaussian filter specifying its length and $\Sigma$.

# 1 Related works

## 1.1 The Pixel Replication Algorithm

In our implementation of the zooming algorithm we develop the GPU version of the already existing algorithm named “Pixel replication” also known as the “Nearest neighbor interpolation”. 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 in implementing the zooming technique but, on the other hand, as the zooming factor increased the resulting image got more blurred.

## 1.2 Image Filtering Convolution Algorithm

For the purpose of improving the visual quality of the zoomed image we implement an image filtering algorithm, based on the convolution operation, 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, we had to manipulate the already known version of the “Image filtering through convolution” Algorithm. 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 non optimized approach that can be improved, for example, exploiting the GPU parallelism concurrently with the tiling technique.

## 2 Proposed Method

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

- Kernel loading
- Image reading
- Image cut out
- Image enlargement and filling
- Image enhancement

The algorithm is implemented in C++ and CUDA, and it is divided into two main parts: the first one is the CPU part, 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 and filling, and the image 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 $-c$ flag, which chooses a custom kernel, followed by the file path of the kernel;
- The $-g$ flag, which chooses a gaussian kernel, followed by the length of the kernel and the $\Sigma$ value, mutually exclusive with the previous;
- An optional $-v$ flag, which enables the verbose mode;
- An optional $-f$ flag, which forces the use of global memory instead of the shared;
- The coordinated 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, which must be an integer bigger than 0.

### 2.1 Kernel Loading

The algorithm can choose between custom kernels given as input from files, or gaussian ones, which are generated on the fly, taking as input the length of the kernel and the $\Sigma$ value. Once generation is done, the kernel is loaded into the constant memory of the GPU.

## 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 dimension of the cut area is passed through command line and the script subsequently calculates the dimension of image in output.
The measure of the side of the final image is computed choosing the largest multiple, smaller than the smallest dimension between width and height of the original image. These checks are performed in order to avoid the creation of a final image with a dimension that is not a multiple of the side of the cut area.
The function carries out the logic explained down below:

$$img_out[tid] = img[starting\_byte + row\_offset + column\_offset] \tag{1}$$

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, so that all holes are filled.
The holes are filled with the help of the pixel replication algorithm. It replicates the neighboring pixels in order to increase them to enlarge the image: it creates new pixels from the already given ones, and it is the most basic technique of implementing the zooming technique, giving as output an image blurred and with lots of artifacts.
In the first version of the algorithm, while filling the canvas with the enlarged image, a black border is left around the image, which will be used to perform the convolution with the filter, which needs a larger image than the one that is actually returned as output, whichever the position from which to start the zooming from. The final version of the algorithm, instead, performs the enlargement, and if the cutout is not at the border, the program will use neighboring pixels to create a slightly bigger image for the convolution. If the cutout is at the border, instead, the back border will be used just for the border part, leaving a small, unnoticeable shade at the last pixels of the image. This final version allows to have an image which is even more faithful to the original one.

## 2.4 Image Enhancement

This step is based on the convolution operation which is characterized by the type of the odd-length kernel mask used and the memory management configuration set in order to divide the main image into smaller pieces by which divide the work load.

The Gaussian kernel mask is one of the two options available in the project, generated on CPU according to the *Gaussian_Kernel_Cpu* function: it receives the length and the sigma of the kernel and calculates the mask based on the below given Gaussian Distribution. 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. It is also possible to utilize a custom kernel passed through the command line which has to be specified element by element and it has to be at maximum *MAX_KERNEL_DIM* long: it is set to 127 elements. The chosen mask is stored in the *d_kernel* parameter allocated in the constant memory: having the mask in the constant memory allows us to save time because it 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.

The convolution operation can be implemented using exclusively the global memory but in some cases, checked in the main file, it can be optimized by exploiting the shared memory through the tiling process. The global memory convolution version is implemented in the *globalCudaUpscaling* function and the shared memory one, when available, is performed in the *tilingCudaUpscaling* function: it is also possible to force the global memory version through the boolean variable *forceGlobal* that can be set in the main call. The image is fragmented in order to work on its multiple sections in parallel where each image-fragment is processed in groups of three blocks: one for each color. Every block of threads only works on a specific color channel and the convolution is performed between the individual color channel of the single pixel and the mask. Within the GPU kernel call it has to specified the shared memory allocation dimension: there must be a specific shared memory chunk for every block of threads. The shared memory has to be first filled with the color information of the image and than it can be used to perform the convolution between the informations stored and the mask: all the 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 $bigTileDim * bigTileDim$.

## 3 Experimental Results

## 4 Conclusions

In this paper we proposed a technique for both zooming and enhancing a digital picture, in RGB colors, by exploiting the gpu parallelism.

At first we started implementing the CPU version of the algorithm not being able to parallelize anything in the workload and then, moving on, we came up with a first non-optimized implementation on the GPU: at the end we succeeded in enhancing the procedure by working on its bottleneck which is the memory reading for the data transfer.

Concerning the employment of the shared memory for the tiling process, we were surprised by the complete absence of memory access conflicts: this led us to the Nvidia developer website where we were pleased to find out that the GPUs equipped with a computer capability greater than the 2.0 version are able to avoid any type of conflict of that kind.

In a future perspective, within the overall Upscaling algorithm implemented, the section concerning the zoom algorithm can certainly be optimized: we utilized a basic algorithm that simply replicates the adjacent pixels but, for example, it could be used another type of algorithm able to interpolate the color values instead of replicating the already existing ones.