Image Zooming and Upscaling Using CUDA

GPU Programming course
A.Y.2022/2023

Politecnico di Torino

1859

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 from 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 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 Σ.

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 to implement zooming but, on the other hand, as the zooming factor increases the resulting image gets 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 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 unoptimized approach that can be improved, for example by 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
- 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 chooses a custom kernel, followed by the file path of the kernel;
- The -*g* (–*gauss*) flag, which chooses 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, which must be an integer bigger than 0, a multiplier of the cut-out area dimension;

```
# Example of command line with gaussian kernel
# cutout center (100,100), zoom 2, dimensions 50x50, kernel size 31, sigma 5
./upsCu -g ./img.ppm 100 100 50 50 2 31 5
# Example of command line with custom kernel
# cutout center (100,100), zoom 2, dimensions 50x50, kernel file ./kernel.txt
./upsCu -c ./img.ppm 100 100 50 50 2 ./kernel.txt
# Example of command line with global memory
# cutout center (100,100), zoom 2, dimensions 50x50, kernel size 31, sigma 5
./upsCu -gf ./img.ppm 100 100 50 50 2 31 5
# 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 Σ value.

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

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, allowing the program to save time 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 the every value of the kernel is read at the same time by every thread of the program, and the kernel 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, so that all holes are filled. 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 of implementing the zooming technique, but it is the most efficient one, since it does not require any computation.
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.

```
...
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, 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.

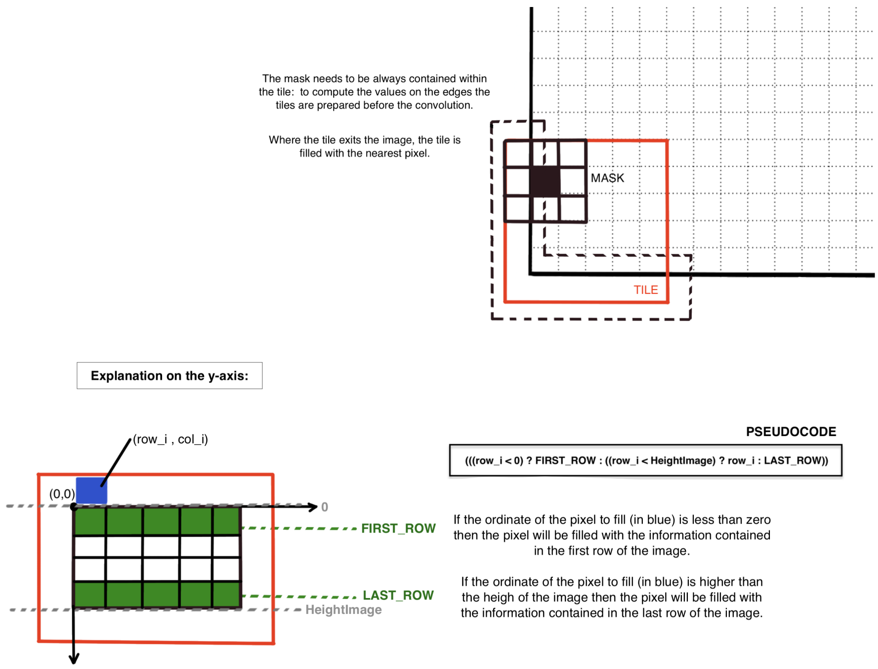

Figure 1: Border pixel replication (same happens for columns)

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 with a function performing the enhancement using only global memory, and then moving to a function using shared memory and a tiling technique.
The first version of the algorithm, which uses only global memory, performs the operations slower than the other, but it is the most straightforward one, and it is the one that has been used for the first tests. The possibility to use it if the user wants to, is still available. The second version of the algorithm, which uses shared memory, improves the performances of the algorithm, given that the shared memory allows to have a faster access to the data. This happens because it is a cache located on chip, and given that it's accessible by all threads of a block, it provides a mechanism for threads to cooperate, while diminishing the access to global memory, which is in the DRAM. Two main problems arise when using shared memory: the fact that it is limited in size, and it is not possible to use it for all the data that is needed, and bank conflicts, which are caused by the fact that the shared memory is divided into banks, which can be accesses simultaneously by different threads, but only if they are accessing different banks. 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. This is why a tiling technique is used, given that it allows to use the shared memory to its full potential, avoiding conflicts, by using a tiling technique. Tiling is a technique that allows to divide the image in tiles, and to perform the convolution on each of them, then combining the results of the convolutions in order to obtain 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, using a kernel on CPU according to the *Gaussian_Kernel_Cpu* function and stored in the *d_kernel* variable allocated in the constant memory, computes the convolution using one thread for each pixel of the image, dividing the image in blocks all of the same size, excluding the last, which can be smaller. The convolution is performed both per color channel and per single block of pixels of the same dimension of the kernel, centered in the

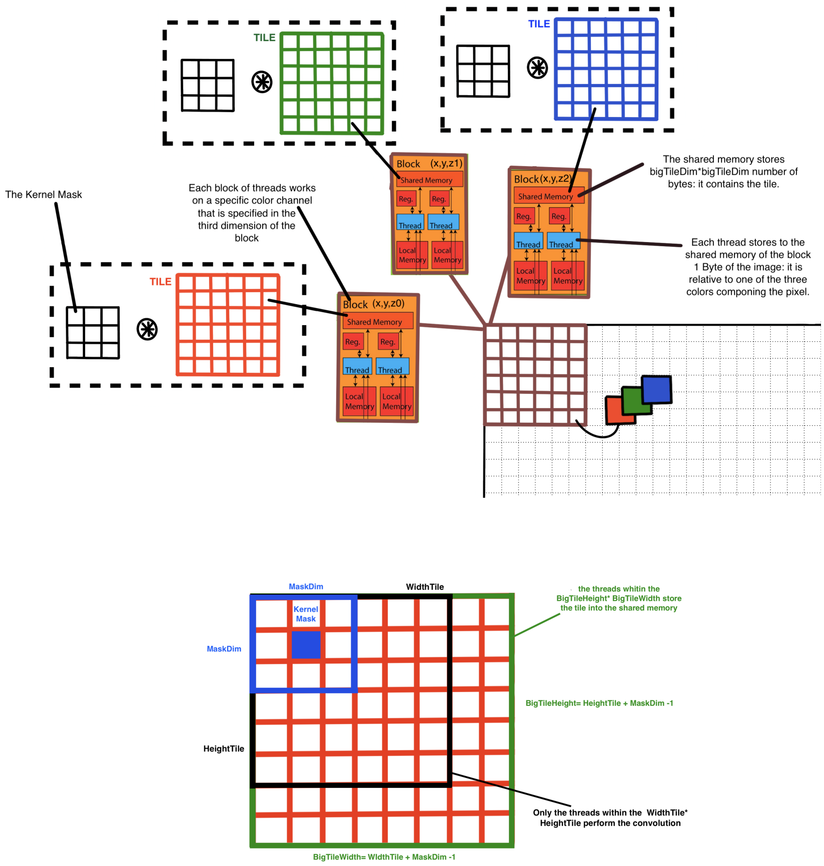

Figure 2: Schematic representation of enhancement process

pixel to be processed. The convolution is performed by multiplying the value of the pixel with the value of the corresponding element of the kernel, and then summing all the results:

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

$$widthTile = \sqrt{maxThreadsPerBlock} - (maskDim - 1) \tag{2}$$

If this value is greater than 1, the tiling technique is used, 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 consideration 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 the cut, the enlargement and the convolution in three different CUDA functions, but this was not the most efficient way of doing it.
The final version, instead, performs the whole process in a single CUDA function, which is called by the main function, and it is the one that is actually executed. This function is called *globalCudaUpscaling* in case the global memory version is used, and *tilingCudaUpscaling* in case the shared memory version is possible.

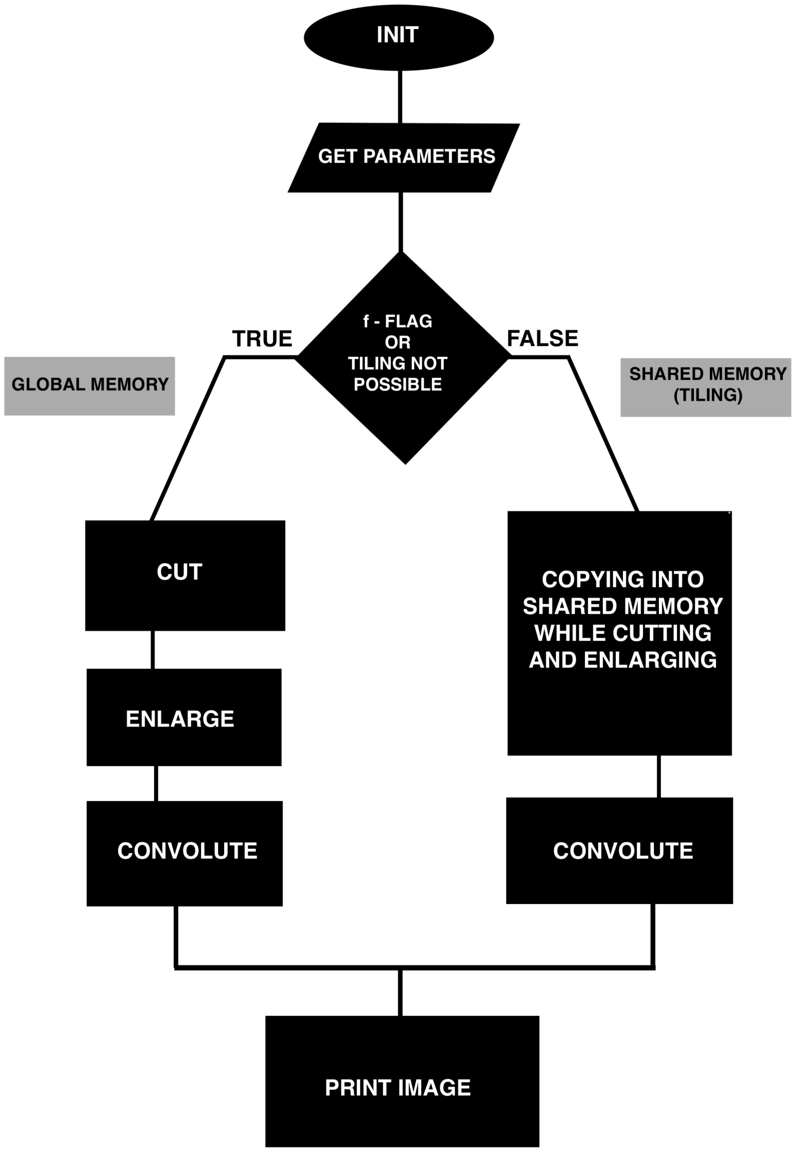

Figure 3: Flowchart

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. 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 18.04.3 LTS and the CUDA version is 10.0.326.

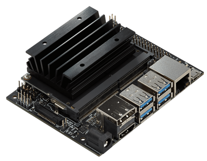

Figure 4: Jetson Nano

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 Changes

The project has been developed in a versioned style. 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 time of the program, since the image is copied three times in the GPU memory, and for each operation the image is copied back to the CPU memory, other than the overhead of the CUDA function calls and kernel creation. Moreover, this version of the project still implements the black border around the image, which creates a worse looking image when the borders are enlarged. In this version, the enlargement of the image must be square, since the complete zooming feature has not been implemented yet.

The second version of the project ($v0.9$) has been developed on January 26th. This version 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, this version of the project implements a complete zooming feature, which allows the user to zoom in the image, by selecting a rectangular area of the image.

The third and final version of the project ($v1$) has been developed on February 10th. This can be considered the final version of the project, since it implements all the features that were planned for the project. This version of the project removes the black border around the image, which makes the image look better when the borders are enlarged, with the technique explained in the previous section.

The three versions have all 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 bring the
These results are presented in the following table 1.

Version	Global or shared memory	Time to execute	Shared ld/st conflicts
v0.5 (19 Jan)	Global	-	/
v0.5 (19 Jan)	Shared	-	0
v0.9 (26 Jan)	Global	-	/
v0.9 (26 Jan)	Shared	-	0
v1 (10 Feb)	Global	-	/
v1 (10 Feb)	Shared	-	0

Table 1: Experimental results

Figure 5: Original image and zoomed image (v1)

4 Conclusions

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

When we first implemented the CPU version of the algorithm parallelization was not achieved in the workload, then 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 bottlenecks like memory reading for the data transfer.

Concerning the employment of the shared memory for the tiling process, we observed complete absence of memory access conflicts: GPUs with a computer capability greater than 2.0 are able to avoid conflicts in our scenario.

From 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. 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.