Politecnico di Torino

GPU Programming

a.y. 2022/23

Pixel art upscaling with xBR

Giuseppe D'Andrea s303378

Mattia De Rosa s303379

Contents

Abstract

This project aims to implement a parallel version of the xBR pixel art upscaling algorithm for GPUs using CUDA API.

xBR ("scale by rules") takes an image as input and returns an upscaled version by a specified factor, usually from 2x to 4x. It is based on pattern recognition and works by comparing the value of each pixel to the surrounding ones, then each pixel is interpolated only if an edge is detected.

The algorithm is designed to work in real time applications like emulators, so our goal is to maximize the performance on the GPU and compare it to a standard CPU implementation.

Problem Introduction

For this project we searched for different algorithms to perform upscaling of pixel-art images. Pixel art graphics have usually very low resolutions and a limited palette of colors, for this reason the traditional scaling algorithms don't' obtain great results. There are different specialized algorithms to solve these problems and are generally used to perform real time upscaling for arcade and console emulators.

The factors that influenced the choice of the algorithm are the overall quality of the result, the possibility to scale the image by different factors and the possibilities of parallelization on a GPU.

As we can see from the image above the bilinear and bicubic filters that work well on general cases, blur the image too much and lose sharpness for the edges making the basic nearest interpolation a better choice for pixel art over them. Scale2x produces a better result over the nearest interpolation because considers the colors of the 4 surrounding pixels when expanding a pixel. But we found the best choice to be the xBR algorithm, because thanks to the edge detection phase and different levels of interpolation produces the best result among the considered algorithms.

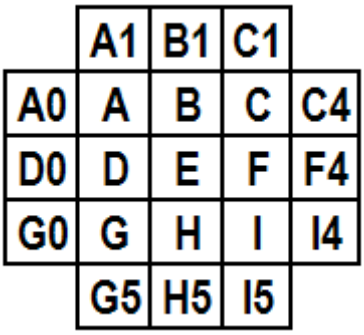
Moreover, it offers a great degree of parallelization because the expansion of a pixel is dependent on the color of the surrounding pixels but independent from their expansion.

In the next chapters we will analyze how the algorithm works and how we implemented it both on CPU and GPU, with focus on different optimizations to increase the performance.

The xBR upscaling algorithm

xBR means “scale by rules” and works by recognizing edges and then interpolating the result in two different stages.

Because of this two-level interpolation it is able handle complex patterns like curves and anti-aliased lines.



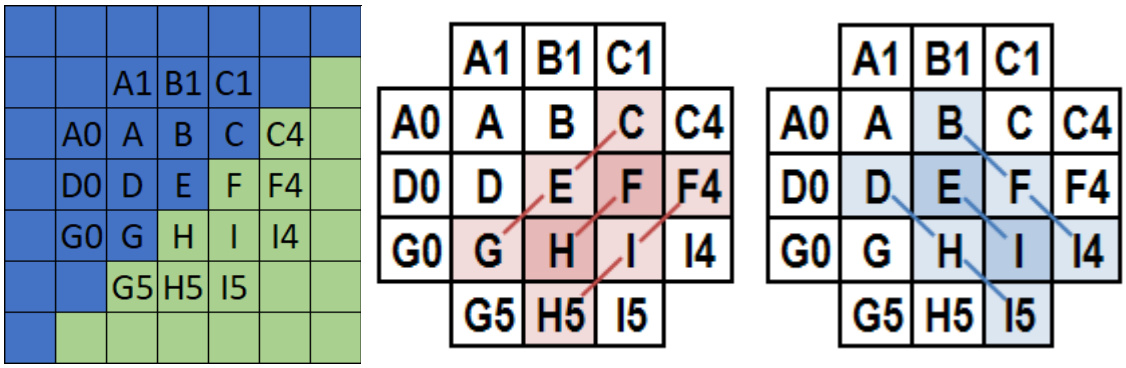
Before introducing the rules we are going to use to perform the upscaling, we have to note that the algorithm has a center symmetry. For brevity the steps are shown only for the bottom right edge, but the rules for the other three edges can be obtained by rotating the pixels by 90°.

Edge Detection Rule

We consider a pixel (E) and its neighbors. The objective is to determine if an edge is present parallel to pixels F-H.

We calculate the weighted color distance between pixels in the F-H (parallel) and E-I (perpendicular) directions.

If the perpendicular color distance is bigger than the parallel one, then it means that an edge is present, and we need to perform the interpolation.



The weighted distance is calculated as follows:

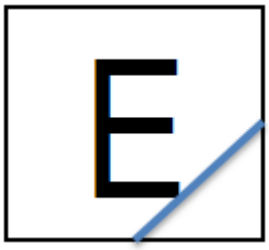
```
wd(red) = d(E,C) + d(E,G) + d(I,F4) + d(I,H5) + 4*d(H,F)
wd(blue)= d(H,D) + d(H,I5) + d(F,I4) + d(F,B) + 4*d(E,I)
edge_present = wd(red) < wd(blue)
```

New Color Rule

We now need to expand the pixel E, depending on the scale factor, we obtain a square matrix with side equal to the scale factor.

If there is no need to perform interpolation then we can simply set the color of all pixels in this output matrix equal to the pixel E, otherwise we need to fill it with different colors based on multiple factors.

From the image on the right we can see where the edge should be placed on the pixel E and the two different regions of colors we need to fill.



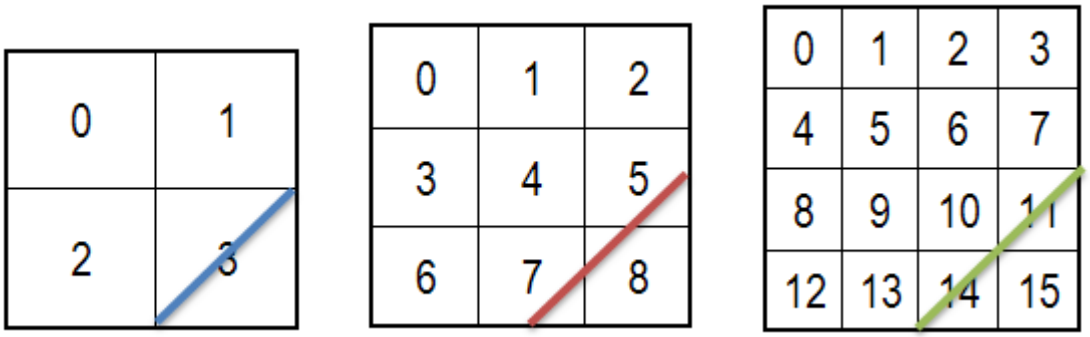
The bigger part needs to be filled with the original color of the pixel E.

The smaller part on the bottom right, needs to be filled by either the color of pixel F or H, according to the following rule:

```
new_color = d(E,F) <= d(E,H) ? F : H
```

Interpolation Level 1 Rule

The first level of interpolation is very basic and can only address 45° edges. Below we can see how the edge goes over a different number of pixels for 2x, 3x and 4x scale factors respectively, but this could be extended for any scale factor.



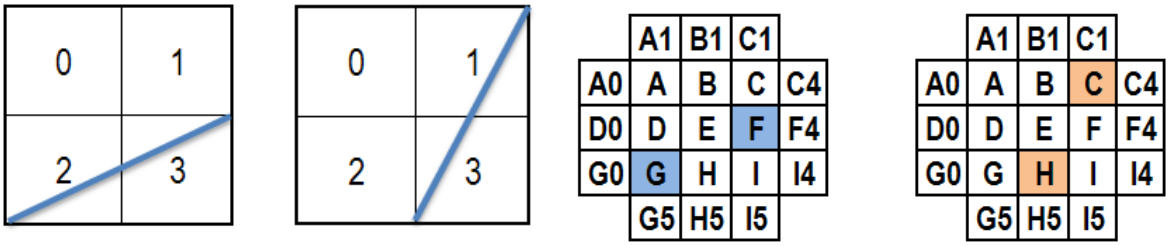
The pixels fully above the line need to be filled with the original color, those fully under it need to be filled with the new color. Finally for the pixels across the edge line we need to mix a new color based on the size of the area on either side of the line.

For example, in the 4x scale example, pixel 15 uses the new color, pixels 11 and 14 use a 50% mix between the original and the new color, while all other pixels use the original color.

Interpolation Level 2 Rule

The second level is more complicated, and is useful to smooth edges that aren't exactly 45° and improves the results on rounded lines.

There are two independent rules, one for the bottom part and the other for the right and it uses the same color strategy used for the first level but with different regions illustrated below.



The rules are as follows:

```
do_LVL2_bottom = (F==G) ? true : false
do_LVL2_right  = (H==C) ? true : false
```

Interpolation Level 2 Issues

We found out that, using a strict equality comparison between the pixel colors, the level two interpolation did not work properly if the image had gradients of colors or if the file used an encoding with compression.

By testing multiple images and videos we found out that png images with no compression worked as expected, but when trying video files the interpolation level 2 was rarely used.

Because of these reasons we decided to check if the color distance between the two pixels was under a specified threshold which depends on the encoding of the input file.

Color Distance

The distance between the colors of two pixels is calculated using YUV color values and uses the same parameters as HQx filters:

```
d(a, b) = 48 * abs(a.Y - b.Y) + 7 * abs(a.U - b.U) + 6 * abs(a.V - b.V)
```

CPU Implementation

We started by doing a simple naïve implementation on the CPU, there is no parallelization of any kind and pixels are processed sequentially. The core of the algorithm is implemented in the `expand_pixel` function which performs the edge detection and interpolation rules.

GPUv0 Implementation

After doing some initial tests on the CPU implementation to check if the algorithm worked properly we created a naïve GPU implementation.

Here we simply turned the `expand_pixel` function into a kernel. We decided to maximize the number of threads per block to 32x32 for a total of 1024 threads per block.

GPUv1

After doing some tests on GPUv0 we quickly realized that we did not get the performance improvement we expected, so by using the Nvidia profiler we found multiple issues impacting the performance.

Floating point precision

Most of the algorithm uses `unsigned char` to perform computation, since the most called function is the distance function between the colors of two pixels. But for the interpolation rules, the function `mix_colors` is called many times and it makes use of a floating point number.

Initially we used a double precision floating point, but quickly realized that it was computationally very heavy. By looking at the technical specification for the jetson nano we can see that the GPU can reach about 7 GFLOPS for double precision floating point, which is a lot lower than the 236 GFLOPS for single precision.

By simply changing the precision used in that computation to 32 bits (single) we were able to considerably reduce the kernel time without impacting the quality of the resulting image.

We tried to reduce the precision further to 16 bits by using `cuda_fp16.h` but we did not see any noticeable improvement even if the GPU should be able to reach 472 GFLOPS for half precision operations. We think this is due to other limiting factors, like memory access time.

	Double precision	Single precision	Half precision
Average kernel time	19.78 ms	13.98 ms	13.14 ms
Double-precision Utilization	High (9)	Idle (0)	Idle (0)
Single-precision Utilization	Low (1)	Low (3)	Low (3)

Tiling

The biggest change in GPUv1 was the introduction of shared memory and tiling the input image.

For the edge detection rule, we always consider the surrounding 5x5 pixels, independently of the scale factor, so we implemented tiling with blocks of size 32x32, a halo of 2 pixels and the actual size of the tile of 28x28 pixels.

We moved the two input arrays (RGB and YUV version of the same input image) to shared memory since every call of the kernel accesses the same pixel multiple times.

After starting the kernel with block size 32x32, every thread copies the value of his own pixel, if it is part of the tile, or generates the color from the nearest pixel if it is in the halo.

After loading the data in the shared memory, the threads are synchronized and 28x28 threads execute the `expand_pixel_tiling` function, which works mostly like the original `expand_pixel` used in the previous versions, except now we use different indices for the input and output matrices.

An additional consequence of tiling was the possibility to remove the YUV image data from global memory and move the conversion from RGB to YUV from its own kernel to `expand_pixel_kernel` which slightly improved the performance thanks to the fewer global memory accesses.

Memory alignment

Tiling proved to be an effective solution to improve the performance of this algorithm, but again by looking at the output of the Nvidia profiler, we found out that there were many more memory accesses than what we predicted.

Now it is time to talk about how we stored pixel data in the memory, we created two c `struct PixelRGB` and `PixelYUV`, both of them contain three `unsigned char` for the three color channels for a total of three bytes.

This is a problem because it causes bank conflicts when multiple threads try to access the same memory. We solved this by aligning the structs to 4 bytes with the directive `__align__(4)`.

Because of this change we had to convert the input data, which was only 3 bytes to the new data structure, since a simple cast was no longer possible. We used another kernel to speed up this operation because it was faster than using the CPU functions provided by the OpenCV library.

The same had to be repeated after the image was scaled, before being able to output it.

The execution of these two additional kernels increased the total execution time, but we still obtained a net gain from solving the bank conflicts.

User Manual

Compile

To read image and video files the program use opencv4 library, so it is needed to include it when compiling using `` `pkg-config --libs --cflags opencv4` ``. Moreover on the Jetson Nano we are limited to 32 registers per block, so we need to limit the number of register with the compiler flag `-maxrregcount 32`.

The complete command used to compile the different versions is:

```
nvcc `pkg-config --libs --cflags opencv4` -lX11 -maxrregcount 32 -O3 main.cu -o main.out
```

Run

Once the selected version is compiled, there are different parameters for the executable:

- Scale factor
- Input type. It could be `i` for image or `v` for video
- Input file
- Output file

Example:

```
./xBR_GPUv1 4 i input.png output.png
```

```
./xBR_GPUv1 4 v input.avi output.avi
```

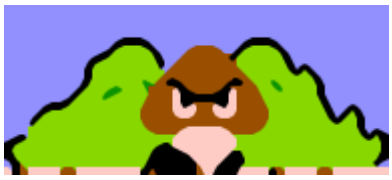
Results

The algorithm turned out to work well for pixel art images, upscaling the original image and smoothing the edges without blurring the result.

Here we present some screenshots taken with an emulator of the Nintendo NES of the game Super Mario Bros. The first image in the triplets is the original from the game, the second one is upscaled using Nearest Neighbors and the third one is upscaled using our implementation of xBR.

1 - Original

2 - Nearest Neighbors 4x



3 - xBR 4x

4 - Original

5 - Nearest Neighbors 4x

6 - xBR 4x

7 - Original

8 - Nearest Neighbors 4x

9 - xBR 4x

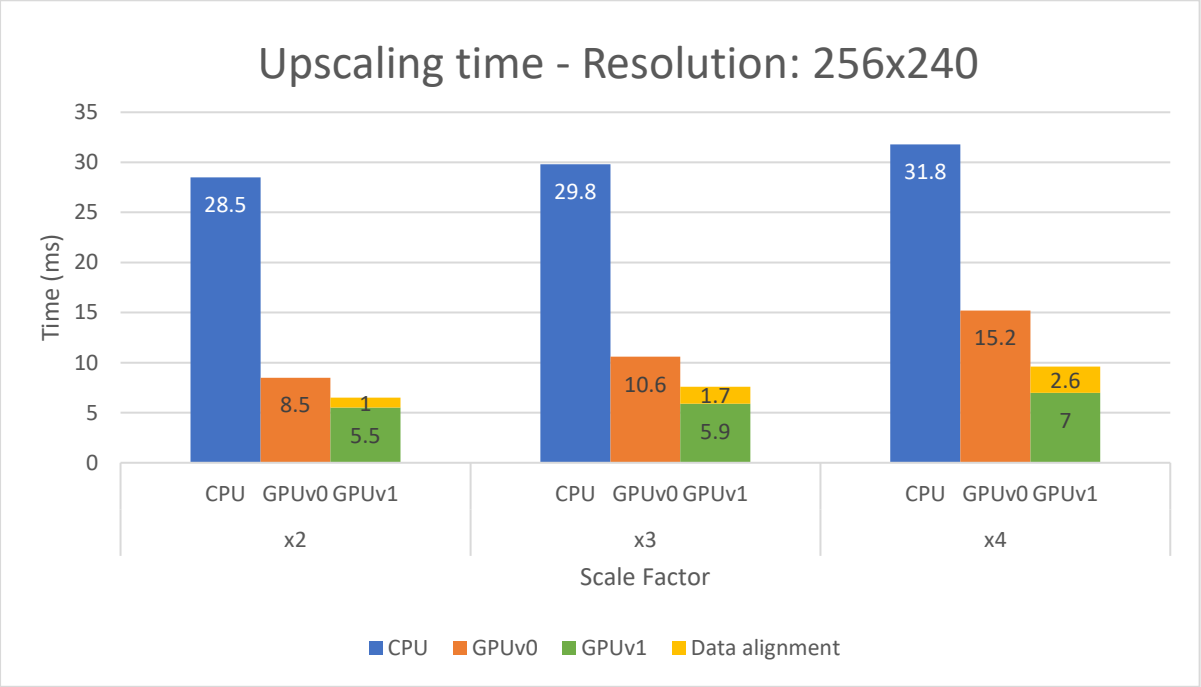
10 - Original

11 - Nearest Neighbors 4x

12 - xBR 4x

Performance

The time evaluated includes the RGB to YUV conversion and upscaling of a single frame and is the average of the results of thousands of run of the algorithm.



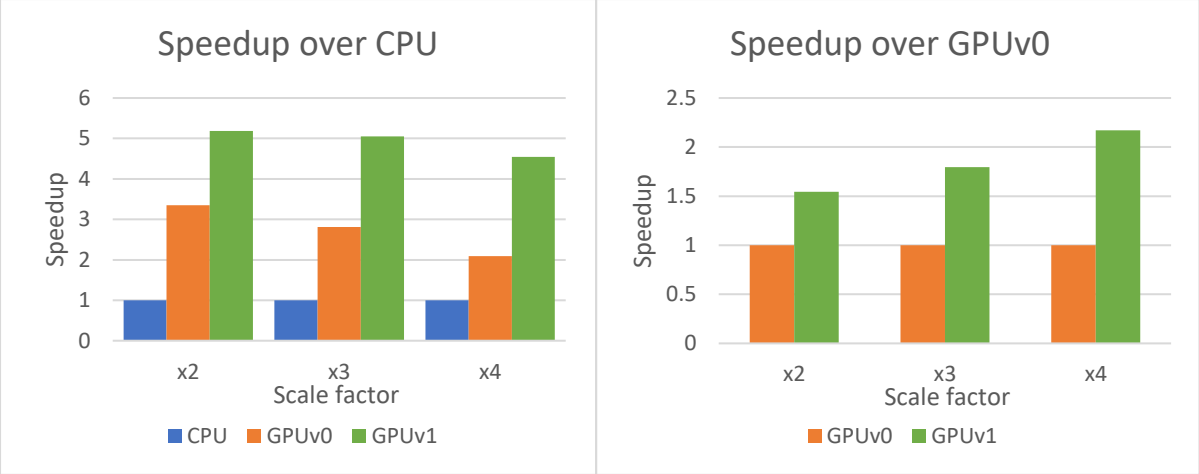
In the above chart we compared the execution time of the expand pixel function (or kernel) in the different versions we developed.

During these tests we excluded the time needed to read and write the image on disk to focus more on the performance of the algorithm.

As explained before, the CPUv1 needs to realign the input data before and after the call to the kernel, this time was included in the resulting graph and shown under the label “Data alignment” to highlight the time difference.

The need for this additional operation was caused due to a limitation in the OpenCV library that, depending on the codec, did not allow us to read/write RGBA data which would have made the alignment unneeded.

To keep the codec compatibility as wide as possible we had to implement this realignment function, which accounts for up to 27% of the total kernel time depending on the chosen scale factor.



In these two charts we computed the speedup over the CPU and the GPUv0.

As we can see the GPU is considerably faster than the CPU, reaching a 5x speedup in the best case (x2 upscale in GPUv1). We can also observe that the speedup decreases with a greater scale factor for the GPU versions, and we think this is caused by an increase in the warp divergence. Checking the values of warp execution efficiency supports this theory.

	X2	X3	X4
Warp Execution Efficiency	44%	40%	38%

The speedup of GPUv1 over GPUv0 instead increases alongside scale factor because there are more memory loads that benefit from the use of shared memory.

Real time test

To test real time performances and avoid the overhead of reading/writing on files and encoding/decoding the image/video files we implemented a function to grab the content of a window on the screen, perform the upscaling and display the output on another window on the screen. In this case it was no longer needed to realign the memory because the functions used to grab the image and to display the output worked with 4 bytes pixels.

This allowed us to view the output in real time and to try it directly on an emulator to see the results. The outcome was that the game was perfectly playable looking at the upscaled version with minimal lag. The occasional lag spikes are present also in the original emulator window and are caused by the high CPU usage of the emulator.

The next picture shows the results with the window emulator on the left and the upscaled version on the right.

If we were to implement the algorithm directly in the emulator code than we could remove the overhead caused by the screen grab.

13 - Screenshot of the real time setup

Conclusions

To conclude, parallelizing this algorithm provided a great benefit in the performance. The high thread count of the GPU works well with this kind of algorithms where the expansion of each pixel is independent from each other.

The use of shared memory improved the performance even more thanks to the multiple accesses for each thread to the value of the pixels.

The biggest limiting factor to the performance of parallel xBR was the interpolation rule. Since there are many different conditions to check if an interpolation should be applied, we create high warp divergence.

This issue is magnified for greater scale factors due to the increased time spent in divergent branches.

To improve this implementation, we think that custom functions to read and write images and videos could help the general performance to avoid the problem of memory realignment and that maybe the expansion of a pixel could be split in four different kernels, one for each edge to check, to reduce warp divergence.

References

I. Pixel art scaling algorithms - https://en.wikipedia.org/wiki/Pixel-art_scaling_algorithms

II. Scale2x - https://www.scale2x.it/

III. Bilinear filter - https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/interpolation/bilinear-filtering.html

IV. HQx - http://blog.pkh.me/p/19-butchering-hqx-scaling-filters.html

V. xBR - https://forums.libretro.com/t/xbr-algorithm-tutorial/123

VI. xBR - http://git.videolan.org/gitweb.cgi/ffmpeg.git/?p=ffmpeg.git;a=blob;f=libavfilter/vf_xbr.c;h=5c14565b3a03f66f1e0296623dc91373aeac1ed0;hb=HEAD

VII. CUDA Shared Memory - https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

VIII. Jetson Nano Technical Specifications - https://www.techpowerup.com/gpu-specs/jetson-nano-gpu.c3643