Parallel Computing 2019/2020
Kernel Image Processing
Image blurring through C++11 Multi-Threading parallelization

Andrea Spitaleri
spita90@gmail.com
andrea.spitaleri@stud.unifi.it

Abstract

In this work we will discuss about two different approaches to image editing: Single-Thread Sequential Computing and Multi-Thread Parallel Computing.
In particular, we will use C++11 Multi-Threading for a task such as image blurring, an we will then evaluate the obtained results.

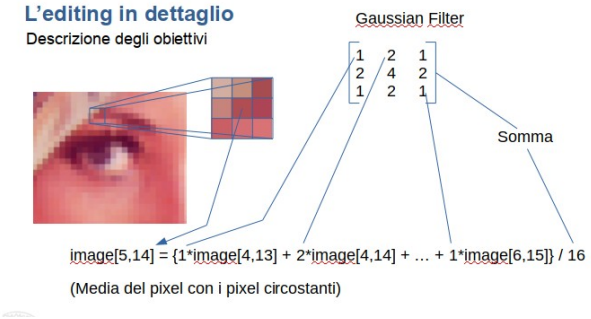
1. Overview

We want to blur a given image. This is obtained, in most software, by applying a convolution mask to all pixels.

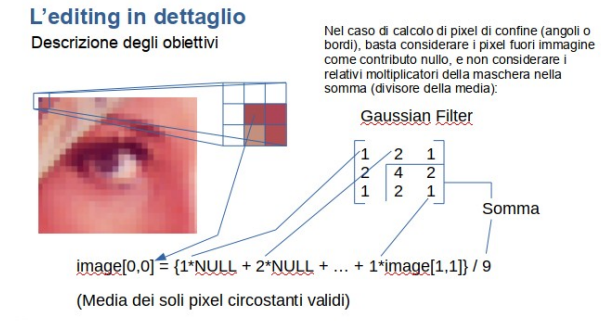
In a Single-Threaded approach, the software computes all pixels in a sequential manner, while in the Multi-Threaded approach, adopted in my software, the image is divided into portions, which are then given to the CPU cores, so each core can compute its portion in parallel manner.

2. The editing phase in detail

The convolution mask is just a way to give a pixel the mean of the colors of all surrounding pixels. This is applied to all pixels in the image.



If we want to compute the color of a pixel at the border of the image, we should only consider inner pixels.



3. The problems in Multi-Thread computation

In the Single-Threaded scenario, all image pixels are computed by a single compute unit.

If we want instead, to use all available compute unit, we have to split the workload so we can exploit parallel computing.

This subdivision however makes us have to deal with some technical details that we should care about.

First of all: how to split the image in a proper way?

4. The two different approaches

We can think about two different approaches: one is to divide the image into fixed size blocks, and another is to use, instead, dynamic sized blocks.

Let us see what the first approach is about.

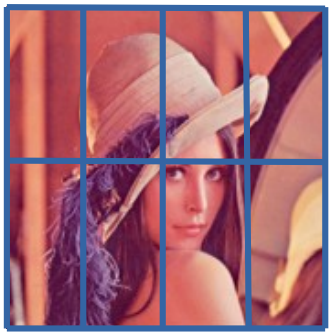
Fixed sized blocks is a possible way to subdivide the image, but it can introduce some critical details:

- The image could not fit precisely into the blocks grid (like in the image shown above). Some blocks (those at the bottom and those at the right side) will then waste some time doing useless work.
- Depending on image resolution and block size we could have a lot of overhead caused by thread scheduling across the various CPU cores.

We would like to have an approach without all this hassles.

Let's see, then, what the second approach (adopted in my software) is about.

In the dynamic sized block approach we have something like this (in an 8 thread example):



It solves that problems, but what happens if the image is not squared, like in this case, but instead it is in landscape or portrait mode?

How many subdivisions should we do in the horizontal and in the vertical directions?

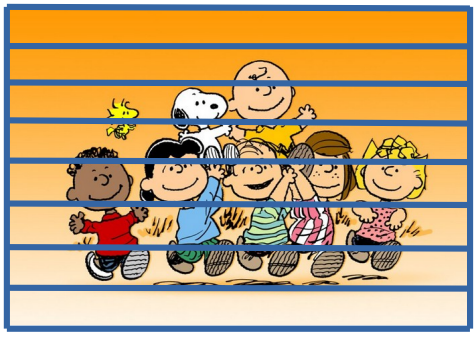
500x300 image

In the 8 core landscape scenario we can see that in a vertical-only subdivision, each thread needs, other than the block it is working on, a border of pixels (for the computing of the borderline pixels color) made by two 500/8 pixels segments and by two 300 pixels segments. So, the additional border part is, for each thread, made by 725 pixels.

For the first and the last thread 300+2*(500/8) pixels are null, because they are outside of the image, and only 300 pixels are "true" pixels. For the intermediate threads, instead, only 2*(500/8) pixels are null.

If we ignore null pixels, we have in total, for all 8 threads: 6*(2*300)+2*(300) = 4200 additional pixel computed.

500x300 image



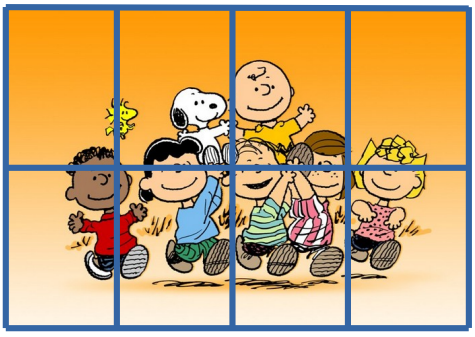
In a horizontal-only subdivision, each thread needs, other than the block it is working on, a border of pixels (for the computing of the borderline pixels color) made by two 500 pixels segments and by two 300/8 pixels segments.
So, the additional border part is, for each thread, made by 1075 pixels.
For the first and the last thread 500+2*(300/8) pixels are null, because they are outside of the image, and only 500 pixels are “true” pixels. For the intermediate threads, instead, only 2*(300/8) pixels are null.

If we ignore null pixels, we have in total, for all 8 threads: 6*(2*500)+2*(500) = 7000 additional pixel computed.

This is much worst than the previous case, but only for landscape images! It is actually the opposite for portrait images!

So we need a universally good way to split the image.

500x300 image



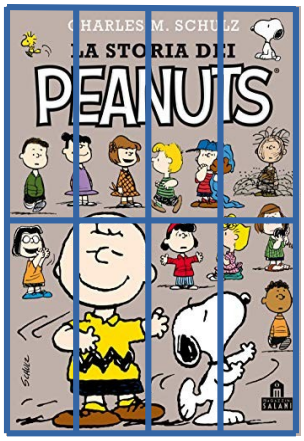
Mixed subdivision: each thread needs, other than the block it is working on, a border of pixels (for the computing of the borderline pixels color) made by two 500/4 pixels segments and by two 300/2 pixels segments.
So, the additional border part is, for each thread, made by 550 pixels.

For the corner threads, half pixels are null, because they are outside of the image, and only the other half are “true” pixels.
For the inner threads, instead, only 500/4 pixels are null.

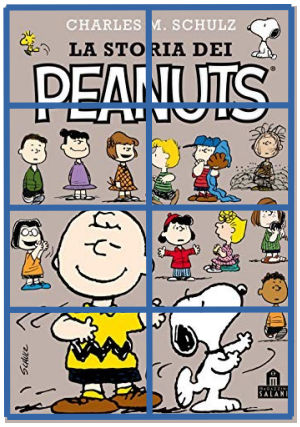
If we ignore null pixels, we have in total, for all 8 threads: 4*(500/4+300/2)+4*(500/4+2*(300/2)) = 2800 additional pixel computed.
Now we’re talking!

Let´s see now what we’d have with portrait images.
Should we subdivide more on the short side or more on the long side?

300x500 image



If we ignore null pixels, we have in total, for all 8 threads: 4*(500/2+300/4)+4*(300/4+2*(500/2)) = 3600 additional pixel computed.



If we ignore null pixels, we have in total, for all 8 threads:4*(500/4+300/2)+4*(500/4+2*(300/2)) = 2800 additional pixel computed.
This is the case we saw earlier.

So it seems clear that it’s always better to make more subdivisions on the long side wrt the short side.
For this reason, my software implementation preventively

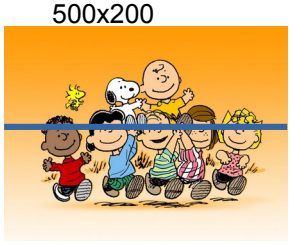
checks if the input image is in landscape or portrait mode.

5. Image subdivision wrt thread number

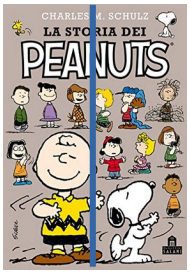
Now that we saw that it is more convenient to split more on the long side wrt the short side, let's see how this result may vary by varying thread number.

2 Threads

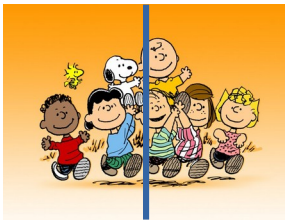
500x200

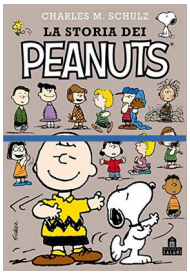


200x500



Short-side subdivision: in both cases we have for each block 2*500 = 1000 additional pixel computed.

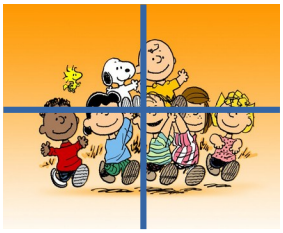




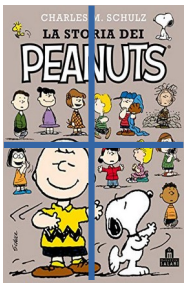
Long-side subdivision: in both cases we have for each block 2*200 = 400 additional pixel computed.

4 Threads

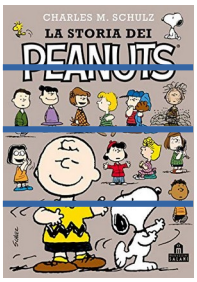
500x200



200x500



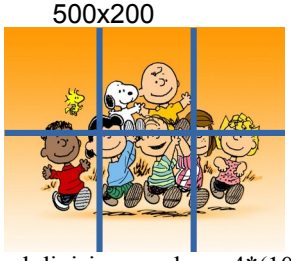
So, we stop subdividing on the short side.
Mixed subdivision: we have 4*(250+100) = 1400 additional pixel computed.



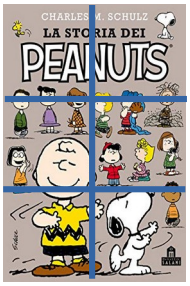
Long-side subdivision: we have 2*200+2*(2*200) = 1200 additional pixel computed.

6 Thread

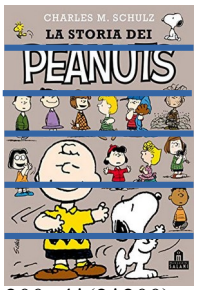
500x200



200x500



Mixed subdivision: we have 4*(100+167)+2*(2*100+167) = 1802 additional pixel computed.

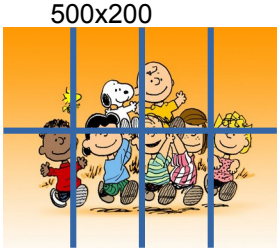


Long-side subdivision: we have 2*200+4*(2*200) = 2000 additional pixel computed.

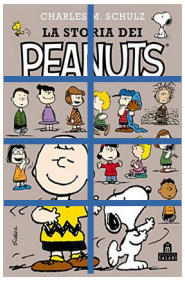
So, for more than 6 threads it seems that it is not convenient anymore to split only on the long side.

8 Thread

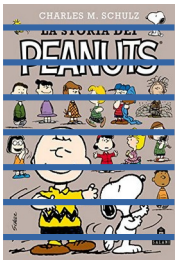
500x200



200x500



Mixed subdivision: we have 4*(100+125)+4*(2*100+125) = 2200 additional pixel computed.



Long-side subdivision: we have 2*200+6*(2*200) = 2800 additional pixel computed.

This confirms the fact that from 6 threads up, it is convenient to split also the short side.
All right, but how many subdivisions should we do to the short side? Well, we can see that it is bad to split the short side more than once, because this will create inner blocks with absolutely zero null pixels!

6. Image editing performance analysis

Using the timers integrated in the software GUI, we can look at the speedups that we can obtain by varying some parameters such as number of blurring passes, image resolution, and number of threads.

The following benchmarks are done with this system:

Laptop Xiaomi Mi Notebook Pro
CPU Intel Core i7-8550U (8 logic cores @ 1.8 - till 4 GHz in Turbo Boost)
RAM 16 GB DDR4
O.S. Manjaro Linux 18.1.4 (based on Arch Linux) with Linux Kernel 5.4

We will take 5 times for each case, and the times reported on the following tables are the mean times.

100x100 image

BOX	*Risoluzione*					
	100x100					
	Passaggi					
	10		50		100	
Thread	*Tempi (ms)*					
1 (Single)	34		137		238	
	Tempi (ms)	*Speedup*	*Tempi (ms)*	*Speedup*	*Tempi (ms)*	*Speedup*
2	13	2,62	54	2,54	86	2,77
8	14	2,43	60	2,28	103	2,31
16	15	2,27	63	2,17	104	2,29
64	25	1,36	75	1,83	136	1,75
100	37	0,92	92	1,49	228	1,04

We can see that for very small images, we have the best results, however small, with only 2 threads. A little worst with 8 threads, and more and more worst result by increasing thread number. This is understandable and reasonable, since the overhead caused by thread scheduling increases more and more, letting the Multi-Threaded approach become inappropriate.
We can also see that for very small images, increasing the number of blurring passes does not affect speedup.

640x480 image

BOX	*Risoluzione*					
	640x480					
	Passaggi					
	10		50		100	
Thread	*Tempi (ms)*					
1 (Single)	751		3663		7310	
	Tempi (ms)	*Speedup*	*Tempi (ms)*	*Speedup*	*Tempi (ms)*	*Speedup*
2	385	1,95	1891	1,94	3688	1,98
8	212	3,54	1100	3,33	1970	3,71
16	236	3,18	1080	3,39	2014	3,63
64	228	3,29	1073	3,41	2047	3,57
100	224	3,35	1111	3,30	2081	3,51

For medium-sized images the situation changes a lot.
We, indeed, have not just better speedup values, but also we can see that these speedups are gained with a greater number of threads. In fact, for these images, for a small number of threads we have low speedup.
We can also see that for medium-sized images, increasing the number of blurring passes affects speedup just by a little bit.

1920x1080 image

BOX	Risoluzione					
	1920x1080					
	Passaggi					
	10		50		100	
Thread	Tempi (ms)					
1 (Single)	5097		26321		51381	
	Tempi (ms)	Speedup	Tempi (ms)	Speedup	Tempi (ms)	Speedup
2	2606	1,96	15005	1,75	30079	1,71
8	1326	3,84	9229	2,85	19883	2,58
16	1341	3,80	8128	3,24	20185	2,55
64	1324	3,85	9510	2,77	19077	2,69
100	1328	3,84	10255	2,57	20100	2,56

For high resolution images we have a speedup trend, in function of the number of threads, that is quite similar to the medium-sized image scenario. So, it seems that by increasing image resolution, the “right” amount of threads to get best speedups tends to coincide with the number of CPU cores.
We can also see that for high resolution images, increasing the number of blurring passes affects speedup a little bit more wrt previous cases.

7. Final overview

We saw that, in an image editing scenario, a CPU with many cores is more and more needed, since we can have better speedups with a number of threads similar to the number of available CPU cores.

This fact mirrors the rality in a perfect way, since all image editing professionists want to use PCs with many and may cores.

Andrea Spitaleri