UNIVERSITÀ DEGLI STUDI FIRENZE

Parallelization of 2D Images Convolution algorithm

Salvatore Cerullo

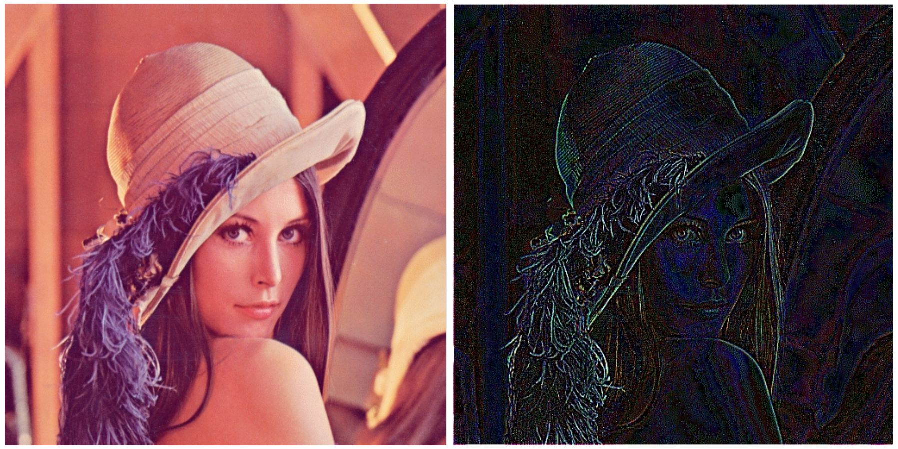
Florence, 10/02/2019

Introduction to Digital Image Processing

There are many *techniques of transformation of a digital image* that can be realized through two types of operators:

- *Point Operators*: transform the value of a pixel based on the value that only the pixel has in the original image;
- *Spatial Operators*: to determine the value of the destination pixel, in addition to using the pixel itself in the original image, we also need the value of some neighboring pixels.

The **filtering** can be performed in the spatial domain through the convolution using ***Kernel*** (or *Mask*).


Edge detection Filtering

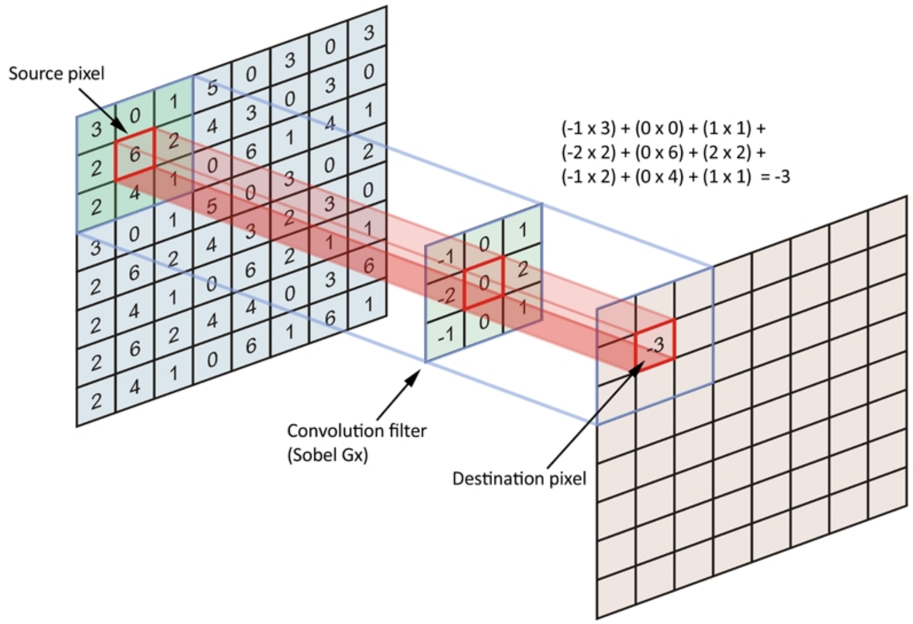
Introduction to Digital Image Processing

2D Image Convolution

A 2D image can be seen as an array of ***n x m*** size.

For convolution of 2D image the kernel must be an array of ***q x q*** size.

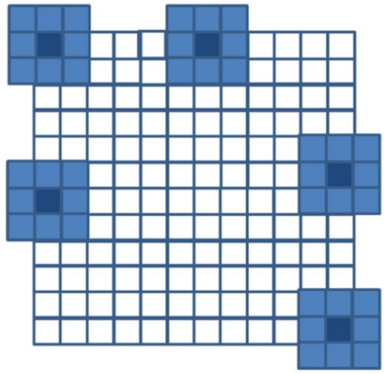
How the convolution work?


Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Introduction to Digital Image Processing

2D Image Convolution

Problem: When overlapping the kernel with pixels along the edges, we notice that the kernel is coming out of the image. How can we calculate the value of the new pixel?



Solutions:

1) We extend the image size by multiplying the value of the kernel with the pixel value of the nearest image;
2) Let's simply consider 0 the value of all the products among the kernel elements that come out of the image.

Implementations

Implementations ***CPU-based***:

• Sequential Algorithm;

• Parallel Algorithm with OpenMp

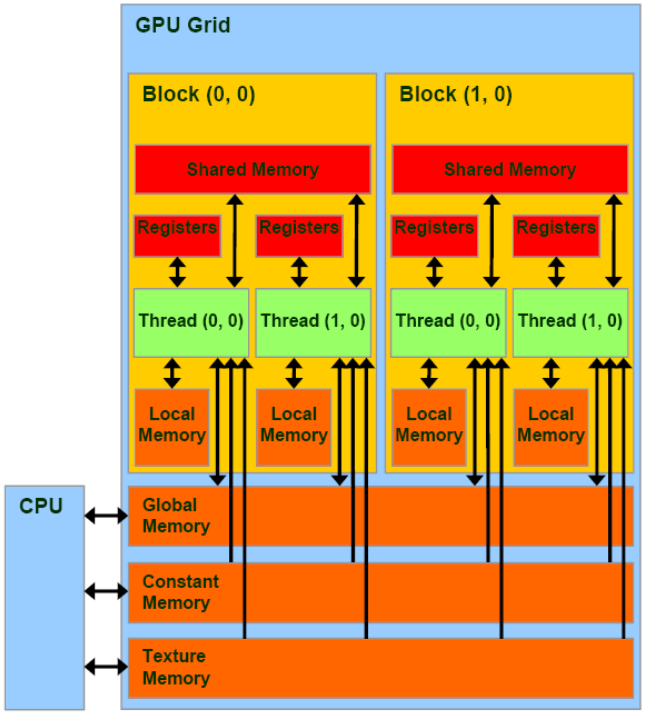
Implementations ***GPU-based***:

•Algorithm using only *Global Memory*;

•Algorithm using *Shared Memory*;

•Algorithm using global Memory and Kernel on *Constant Memory*;

•Algorithm using *Shared memory* and *Kernel* on *Constant Memory*.



Implementations

Sequential Algorithm

The 5 basic steps of this algorithm are:

1) Identification of a single pixel of the original image;

2) Overlay the kernel with the original image, center it in the pixel to be processed, and calculate the products of similar elements;

3) Add the products calculated in the previous point;

4) Insert the calculated value into the output image;

5) Perform steps 1, 2, 3 and 4 for each pixel in the original image;

Implementations

OpenMP

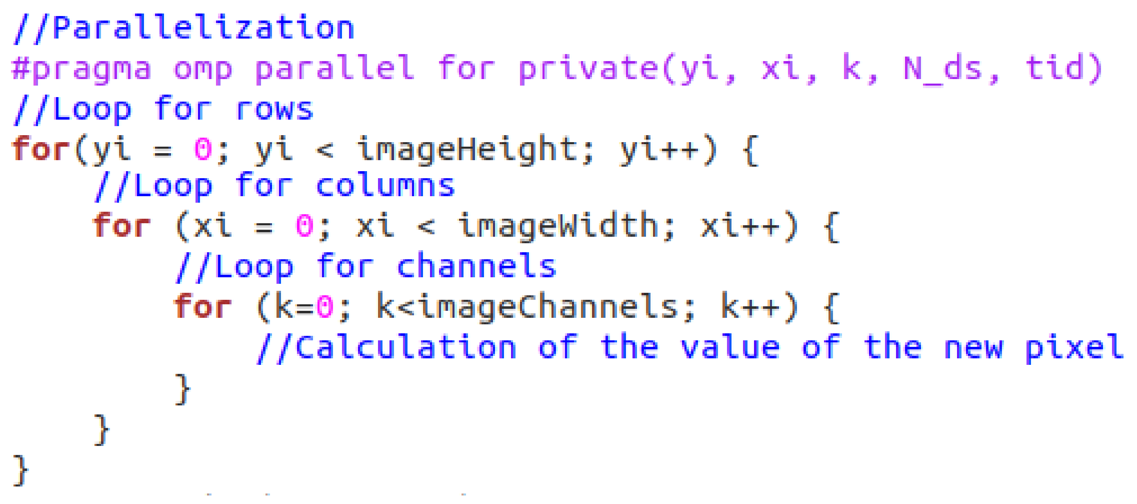
- It is a multiplatform API for creating parallel applications on shared memory systems. It is supported by a different programming languages such as C / C++ and Fortran.
- It is an implementation of the multithreading concept in which a master thread creates a number of slave threads and a process is split between the various slave thread.
- Is based on a combination of directives for the compiler, library functions and environment variables that allow expressing parallelism.

Implementations

Parallel algorithm using OpenMP

It is easy to understand that the sequential algorithm is very expensive and will worsen as images grow larger and larger. The first parallel algorithm is implemented with *C ++* using the *OpenMP API* and will then run on CPU.

The structure of the algorithm is the same as the sequential one but different pixels are processed simultaneously. This is possible thanks to the ***parfor*** construct.



```
//Parallelization
#pragma omp parallel for private(yi, xi, k, N_ds, tid)
//Loop for rows
for(yi = 0; yi < imageHeight; yi++) {
    //Loop for columns
    for (xi = 0; xi < imageWidth; xi++) {
        //Loop for channels
        for (k=0; k<imageChannels; k++) {
            //Calculation of the value of the new pixel
        }
    }
}
```

GPU-based implementations

CUDA

•It is a parallel processing architecture introduced by NVIDIA that allows to significanly increase the calculation performance thanks to the exploitation of the computing power of the GPU (Graphics processing unit).

•It provides some simple extensions that allow you to express parallelism in high-level languages like C,C++, Fortran.

•Supports the joint execution of CPU-GPU. For this reason the code cuda is divided into two parts: the host code is executed on the CPU and the device code on the GPU.

GPU-based implementations

Algorithm using only Global Memory

Like any CUDA program, the simple algorithm is based on the standard ***CUDA structure***:

1) Allocate the memory in the GPU;

2) Copy data from the CPU memory to the GPU memory;

3) Invocation of the CUDA Kernel;

4) Copy data from the GPU memory to the CPU memory;

5) Destroy the memories on the GPU.

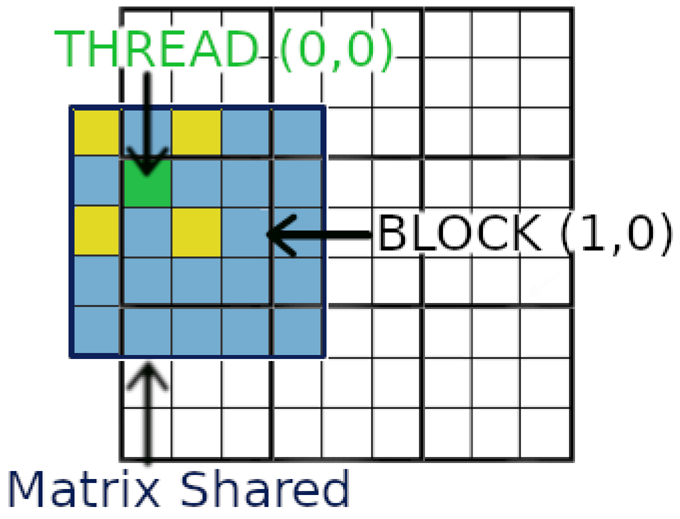
GPU-based implementations

Algorithm using Shared Memory

In the previous approach, each thread reads from the global memory the original image values useful for calculating products with the mask.

With the use of *shared memory* we want to **reduce access** to global memory.

The applied approach exploits the fact that every thread in a block is active. So each block reads 4 pixel values of the original image. In this way we construct a shared array of dimensions
[Block_width + Mask_radius * 2]*[Block_width + Mask_radius * 2].



```
//Each pixel in the image read 4 pixel of the original image
//Case 1: UP_SX
int xx = x - Mask_Radius;
int yy = y - Mask_Radius;
if(xx< 0 || yy < 0){
        N_ds[threadIdx.y][threadIdx.x] = 0;
}else{
        N_ds[threadIdx.y][threadIdx.x] = N[(yy*width+xx)*channels+k];
}

//Case 2: UP_DX
xx = x + Mask_Radius;
yy = y - Mask_Radius;
if(xx>=width-1 || yy < 0){
        N_ds[threadIdx.y][threadIdx.x + 2*Mask_Radius] = 0;
}else{
        N_ds[threadIdx.y][threadIdx.x + 2*Mask_Radius] = N[(yy*width+xx)*channels+k];
}

//Case 3: DOWN_SX
xx = x - Mask_Radius;
yy = y + Mask_Radius;
if(xx<0 || yy > height-1){
        N_ds[threadIdx.y + 2*Mask_Radius][threadIdx.x] = 0;
}else{
        N_ds[threadIdx.y + 2*Mask_Radius][threadIdx.x] = N[(yy*width+xx)*channels+k];
}

//Case 4: DOWN_DX
xx = x + Mask_Radius;
yy = y + Mask_Radius;
if(xx>width-1 || yy > height-1){
        N_ds[threadIdx.y + 2*Mask_Radius][threadIdx.x + 2*Mask_Radius] = 0;
}else{
        N_ds[threadIdx.y + 2*Mask_Radius][threadIdx.x + 2*Mask_Radius] = N[(yy*width+xx)*channels+k];
}

__syncthreads();
```

GPU-based implementations

Algorithms using Constant Memory

Another two variants of the algorithm is implemented thanks to the use of ***constant memory*** that can only be read from the device and then from the kernel.

In order to use the constant memory, the first thing to do is to ***declare a constant variable***:

```
__constant__ float deviceDataMask[Mask_Height * Mask_Width];
```

Next, you need to pass data to the kernel.
To pass data to the kernel, however, you *can not* use the **cudaMalloc** function, but you must use the ***cudaMemcpyToSymbol*** function, as follows:

```
cudaMemcpyToSymbol(deviceDataMask, hostDataMask, sizeAllocMask);
```

In our case we use the constant memory to save the mask data.

The two implementations are:
• *Algorithm using Global and Constant Memory;*
• *Algorithm using Shared and Constant Memory.*

Evaluations

The **test system** was composed of:
- Intel Core i7-3537U CPU;
- 8 GB RAM;
- The GPU was NVIDIA GeForce GT 720M;

The performance evaluations of these algorithms were performed on a set of 7 images:

Name	Dimensions	Size
1.ppm	99x99	29,4 kB
2.ppm	512x512	786,4 kB
3.ppm	1080x720	2,3 MB
4.ppm	1920x1281	7,4 MB
5.ppm	4272x2848	36,5 MB
6.ppm	5472x3648	59,9 MB
7.ppm	6000x4000	72,0 MB

In addition, two kernels of 3x3 and 5x5 sizes were used.

Evaluations

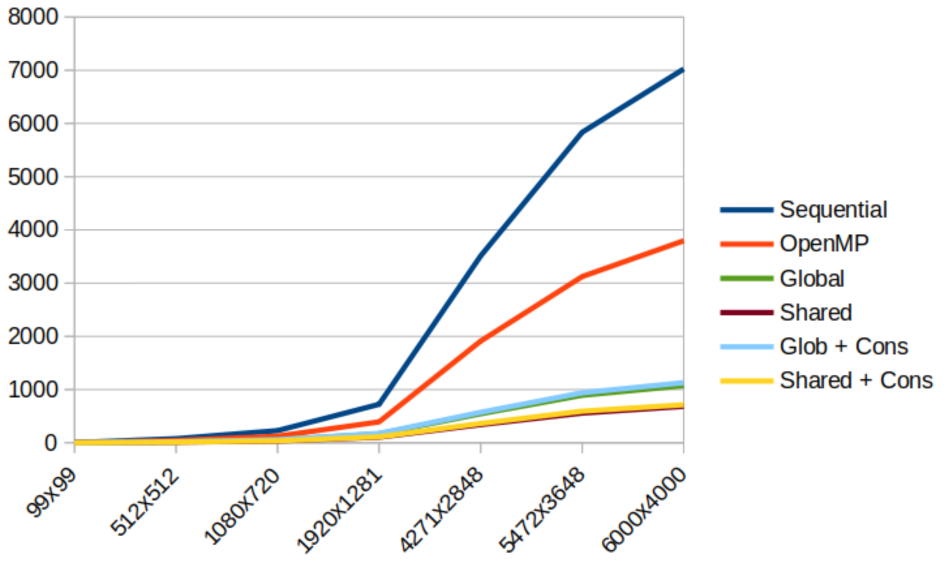
Kernel 3x3

Image	Sequential	OpenMP	Global	Shared	Global + Constant	Shared + Constant
1.ppm	2,88	1,67	0,93	0,58	0,96	0,613
2.ppm	77,28	42,37	17,79	11,07	18,64	11,78
3.ppm	228,41	123,61	52,74	32,91	55,56	35,02
4.ppm	721,83	392,26	166,62	104,23	175,43	110,92
5.ppm	3510,85	1910,18	543,62	339,94	572,49	361,79
6.ppm	5835,63	3122,83	891,47	557,22	938,96	593,16
7.ppm	7023,89	3797,08	1071,7	683,14	1128,66	713,03

Results table for the 3x3 kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection Kernel



Evaluations

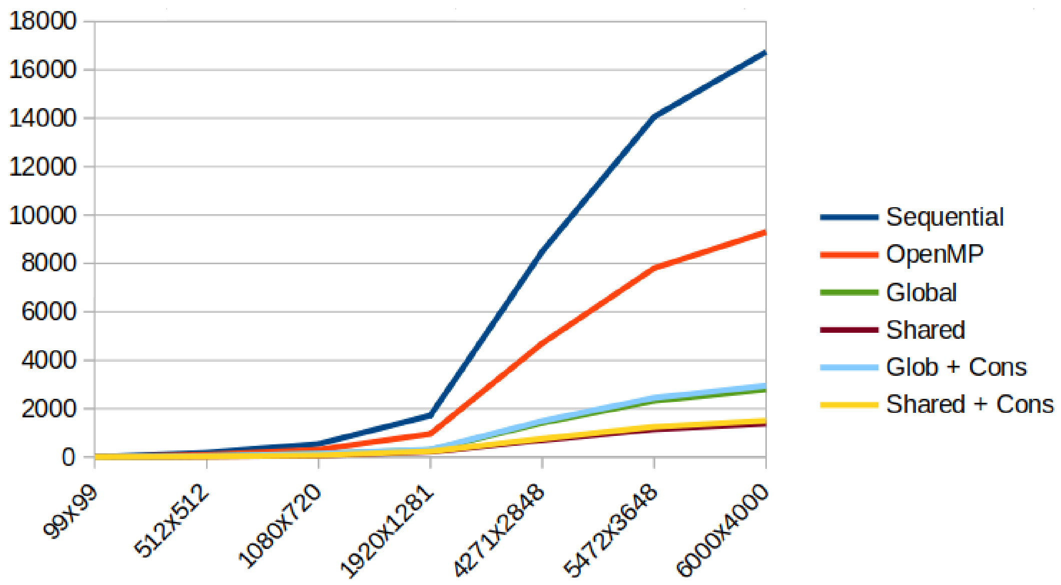
Kernel 5x5

Image	Sequential	OpenMP	Global	Shared	Global + Constant	Shared + Constant
1.ppm	6,81	3,97	1,95	1,15	2,47	1,25
2.ppm	183,03	102,48	46,04	22,71	48,58	24,69
3.ppm	541,57	301,25	137,25	67,71	144,82	73,66
4.ppm	1715,61	958,26	289,06	214,79	305,11	233,74
5.ppm	8497,47	4706,13	1414,91	701,16	1493,22	762,04
6.ppm	14060,13	7803,22	2320,63	1148,43	2449,31	1249,63
7.ppm	16739,62	9298,01	2790,74	1381,23	2944,56	1502,26

Results table for the 5x5 kernel

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Gaussian Blur Kernel



Conclusions

The results obtained show that:

- The sequential version is fast enough for small images;
- The parallel version with OpenMP can halve the time of sequential execution;
- CUDA versions are much more powerful than CPU implementations;
- The best solution is that which exploits shared memory.

Thanks for the attention.