Parallelization of 2D Image Convolution algorithm

Cerullo Salvatore
salvatore.cerullo@stud.unifi.it

Abstract

In digital image processing a very popular operation is filtering in the spatial domain by convolution with specifically designed kernels (filter arrays). These digital filters are usually used for edge detection, sharpen and blur. Nowadays the images are very large and their processing requires great processing capacity. In this article we want to present different digital image convolution algorithms using kernels that exploit parallelism. We will first analyze a CPU-based sequential algorithm and its parallel version. Subsequently we will introduce some parallel versions of the algorithm based on GPU and in particular with the CUDA architecture.

1. Introduction

The Image Convolution is one of the transformation techniques of a digital image also defined as Filtering. There are several operations for processing digital images that use some operators which can be divided into two categories: point operators and space operators. Filtering is part of the local operations, which transform the value of a pixel according to the value of the neighbors pixels, and can be performed through the convolution using Kernel (or Mask), designed specifically to perform a given transformation, such as example the Edge Detection in Fig. 1.

1.1. 2D Image Convolution

The Convolution of 2D iamge consists in recalculating the value of each pixel using the appropriate kernel for the transformation to be applied.

Figure 1. Edge detection filtering.

A kernel is an array of odd sizes such as 3x3, 5x5 and so on.
During kernel processing, the kernel is overlaid for each pixel in the image so that its center matches the pixel to be modified. At this point each element of the kernel is multiplicate with the corresponding element of the original image. After that, all the products calculated in the previous point are added together and the result value is inserted to the new image. The Fig. 2 shows the convolution of a single pixel. This operation must be make for each pixel of the digital image, when each pixel has been processed, the transformation is completed.

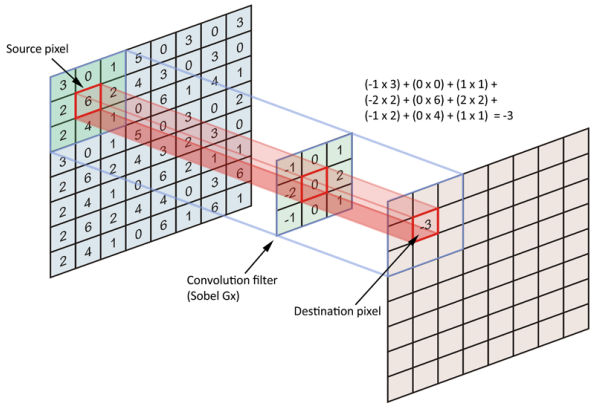

Figure 2. Transformation of a pixel

2. Implementations

In this section we want to present the different implementations of the convolution algorithm of a 2D image that have been realized. The first two algorithms that we are going to present exploit the CPU, the first is a sequential version while the second is a parallel algorithm that uses OpenMP. The subsequent algorithms written in CUDA, on the other hand, are all designed to make the most of the power of the GPU.

2.1. Sequential Convolution Algorithm

As already mentioned, the first algorithm that will be presented is a sequential algorithm that will run on the CPU. In this algorithm, using a series of loops, every single pixel of the original image is sequentially identified and for each of these the new value is calculated by observing the neighboring pixels. The calculation of the new value is strictly dependent on the convolution mask that will be adopted.
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;

2.1.1 Convolution along the edges

A problem that must be managed, in order to carry out the entire operation, is the calculation of the output values for the pixels along the edges. From the Fig 3 we note that when the kernel is centered in one of these points some kernel elements are not overlapping on the image. So how do we calculate the product of these elements? The approach that has been used to eliminate this problem is to set to zero the products of elements that emerge from the image.

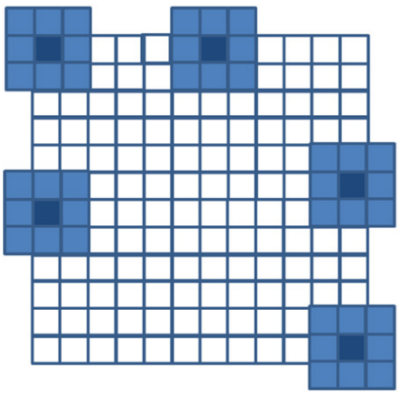

Figure 3. Overlapping the kernel along the edges

2.2. Parallel convolution algorithm on the CPU

It is easy to understand that for a large image the convolution process requires many calculations and consequently a very long processing time. Moreover, observing the algorithm we can see that the Convolution lends itself very well to the Parallel calculation because each pixel is processed independently of the others.
The first parallel algorithm that has been created exploits the CPU again and in particular we are going to use the API for OpenMP programming. The parallelization of this algorithm using OpenMP translates into using the parfor construct on the outermost loop. As we can see from the Figure 4, the construct parfor will be inserted in the outermost section of the different cycles, in our case on the cycle that will flow inside the

```
//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
        }
    }
}
```

Figure 4. Portion of the algorithm code with OpenMP

lines. The choice to put the loop of the rows outside, is due to the fact that within an image there will certainly be many more rows and columns than channels, so to obtain more parallelism the outer loop must be that of the lines or of the columns.

2.3. GPU parallel implementations

After seeing both the sequential and parallel approach for algorithms that exploit the CPU we are going to present some implementations of the algorithm that exploit the computing power of the GPUs.
These algorithms were made using CUDA that is a parallel computing platform and programming model developed by NVIDIA.

2.3.1 CUDA simple convolution algorithm

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 Kernel cuda (shown in Fig. 5);
4. Copy data from the GPU memory to the CPU memory;
5. Destroy the memories on the GPU.

2.3.2 CUDA convolution algorithm using Shared Memory

In the simple algorithm we notice that every thread of a block performs as many accesses to the global memory as there are elements of the mask, for example if we are using a 3x3 mask we will have 9 accesses in global memory. Furthermore, we can see that many threads perform the same read operations as many other threads in the same block.
To eliminate this problem we will use shared memory and the fact that each thread of a block is

```
__global__ void functionConv(float *N, float *M, float *P,
                             int width, int height, int channels){

   float N_ds[Mask_Height][Mask_Width];
      int y = (blockIdx.y * blockDim.y + threadIdx.y);
      int x = (blockIdx.x * blockDim.x + threadIdx.x);

      for(int k=0 ; k<channels;k++){
         for(int ym = 0; ym < Mask_Height; ym++){
            for(int xm = 0 ; xm < Mask_Width; xm++){
                int xx = x - Mask_Radius + xm;
                int yy = y - Mask_Radius + ym;
                if(xx<0 || xx >= width || yy >= height || yy < 0){
                   N_ds[xm][ym] = 0;
                }else{
                   N_ds[xm][ym] = M[ym*Mask_Width+xm]*N[(yy*width+xx)*channels+k];
                }
             }
         }

         float valPi = 0;
            for(int i = 0 ; i < Mask_Height ; i++){
               for(int j = 0 ; j < Mask_Width; j++){
                  valPi += N_ds[i][j];
               }
            }

            if(y < height && x < width){
               P[(y*width+x)*channels + k] = valPi;
            }
      }
}
```

Figure 5. Kernel of simple convolution algorithm

active. Thus, as we can see in the Figure 6, each thread in a block will load 4 pixels in a shared matrix of size $(WidthBlock + MaskRadius * 2)^2$ (larger than the previous approach). After reading the 4 pixels each thread must wait, through the synchronization() method, for all the threads to make the respective readings.
We can therefore say that we have reduced the accesses in Global Memory for each thread to 4 instead of 9 or 25, respectively for 3x3 and 5x5 masks.

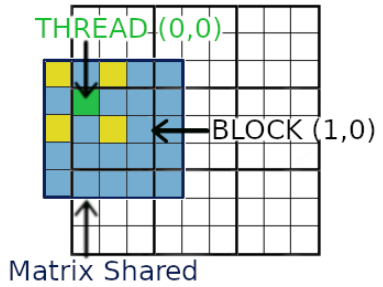

Figure 6. Example of reading in global memory and writing in shared memory of 4 pixels made by the thread (0,0) belonging to the block (1,0).

2.3.3 CUDA convolution algorithms using Constant Memory

A further improvement is achieved thanks to the use of Constant Memory which is used for data that will not change during the execution of a ker-

nel and is read only. Using a constant memory instead of a global one can reduce the bandwidth of the requested memory.
Two algorithms have been implemented that store the mask data within the constant memory. These algorithms are the same as seen previously but with the use of constant memory.
In figs. 7 and 8 we can observe the two instructions that allow us to do this: the first defines the variable as belonging to the constant memory, the second allows us to allocate the space within the constant memory.

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

Figure 7. Array declaration in Constant Memory

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

Figure 8. Copy of the constant data in the gpu

3. Evaluations

The performance evaluations of these algorithms were performed on a set of 7 images with sizes ranging from 99x99 pixels to 6000x4000 and using two kernels of size 3x3 (Sharpen) and 5x5 (Gaussian Blur).
The test system was composed of an Intel Core i7-3537U CPU, 8 GB RAM running Ubuntu 16.04. The GPU was an NVIDIA GeForce GT 720M.
In the tables 1 and 2 the execution times acquired during the tests were reported for each algorithm implemented and performed for the Sharpening operation (kernel 3x3), while in the 3 and 4 tables the results for the operation of Gaussian Blur were reported (5x5).
In CUDA the time is calculated from the kernel call and the allocation and transfer times from the CPU to the GPU memory have not been taken into account.
As far as CPU-based algorithms are concerned, we note that the sequential algorithm is fast enough for very small images, but with increasing size, performance fall. The parallel OpenMP algorithm, on the other hand, optimizes speeds, obtaining almost half the time respect to the sequential algorithm.

Image		CPU	
Name	Resolutions	Sequential	OpenMP
1.ppm	99x99	2,88	1,67
2.ppm	512x512	77,28	42,37
3.ppm	1080x720	228,41	123,61
4.ppm	1920x1281	721,83	392,26
5.ppm	4272x2848	3510,85	1910,18
6.ppm	5472x3648	5835,63	3122,83
7.ppm	6000x4000	7023,89	3797,08

Table 1. CPU-Based algorithms elaboration times (Kernel 3x3)

Image		GPU (CUDA)			
Name	Resolutions	Global	Shared	Glo.+Con.	Sha.+Con.
1.ppm	99x99	0,93	0,58	0,96	0,613
2.ppm	512x512	17,79	11,07	18,64	11,78
3.ppm	1080x720	52,74	32,91	55,56	35,02
4.ppm	1920x1281	166,62	104,23	175,43	110,92
5.ppm	4272x2848	543,62	339,94	572,49	361,79
6.ppm	5472x3648	891,47	557,22	938,96	593,16
7.ppm	6000x4000	1071,70	683,14	1128,66	713,03

Table 2. GPU-Based algorithms elaboration times (Kernel 3x3)

Image		CPU	
Name	Resolutions	Sequential	OpenMP
1.ppm	99x99	6,81	3,97
2.ppm	512x512	183,03	102,48
3.ppm	1080x720	541,57	301,25
4.ppm	1920x1281	1715,61	958,26
5.ppm	4272x2848	8497,47	4706,13
6.ppm	5472x3648	14060,13	7803,22
7.ppm	6000x4000	16739,62	9298,01

Table 3. CPU-Based algorithms elaboration times (Kernel 5x5)

Image		GPU (CUDA)			
Name	Resolutions	Global	Shared	Glo.+Con.	Sha.+Con.
1.ppm	99x99	1,95	1,15	2,47	1,25
2.ppm	512x512	46,04	22,71	48,58	24,69
3.ppm	1080x720	137,25	67,71	144,82	73,66
4.ppm	1920x1281	289,06	214,79	305,11	233,74
5.ppm	4272x2848	1414,91	701,16	1493,22	762,04
6.ppm	5472x3648	2320,63	1148,43	2449,31	1249,63
7.ppm	6000x4000	2790,74	1381,23	2944,56	1502,26

Table 4. GPU-Based algorithms elaboration times (Kernel 5x5)

The algorithms that use CUDA have processing times much lower than the CPU-based algorithms, and we also note that the use of the shared memory further improves performance.