Parallelization of Kernel Processing Algorithm with C++11 and CUDA

Bongini Pietro
pietro.bongini@stud.unifi.it

Abstract

Kernel Processing algorithm, or kernel convolution algorithm is a very used technique in the field of image processing and analysis. It is usually employed for blurring, sharpening, edge detection and more. As this method is widely used and images are increasing in their dimensions, a parallelized approach is needed. In fact, the sequential version could take a long time with a big dataset of images. In this article we present the implementation of the kernel processing algorithm parallelized through C++11 and CUDA technologies.

Future Distribution Permission

1. Introduction

The technique of kernel processing, also known as convolution, consists in the application of a small matrix, named kernel or mask, on a second matrix, the image. Usually kernel dimensions are 3x3, 5x5 and 7x7. The odd dimensions allow to identify the central element of the matrix which is important in the computing phase. In the convolution process, the mask matrix is put on the image matrix so that the center of the mask matches the pixel of the matrix that we want to elaborate. Then the value assigned to this pixel in the output image is computed as the sum of the products of every element of the kernel (with rows and columns flipped) with the corresponding image pixel. This operation is accomplished for all the pixels of the image. Formula 1 shows an example of the convolution's operations with a 3x3 mask for the element at coordinates [2,2].

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} =$$

$$(i*1)+(h*2)+(g*3)+(f*4)+(e*5)+ (d*6)+(c*7)+(b*8)+(a*9) \qquad (1)$$

As we can see a single phase of the convolution requires a lot of calculations. These calculations increase when the kernel dimensions are augmented. For this reason, calculating the convolution for the entire image becomes an extremely onerous operation, and we have to consider some parallelization techniques to improve the performances of this algorithm.
In chapter 2 of this paper we describe the steps of the sequential algorithm. In chapter 3 we present a C++11 parallelized version of the algorithm and then we analyze the results of the experiments. In chapter 4 we present the parallelized version with CUDA and we analyze the results obtained with this last version. Finally, in chapter 5, we draw the conclusions of this project.

2. Sequential Kernel Processing Algorithm

As described before, the algorithm consists in the application of a mask on an image. It performs the following steps:

1. We set the accumulator to zero.
2. We cycle on rows and columns of kernel elements.
3. We multiply each kernel element with the corresponding element of the image and we add the result to the accumulator.

4. At the end of the cycle we assign the accumulator value to the image pixel which corresponds to the central element of the kernel.

5. We repeats 1) 2) 3) 4) for all the pixels of the image.
   An aspect to be taken into account is the way the edge of the image is handled. In fact, when the convolution is computed and the image pixel corresponding to the central element of the kernel is an edge element, there' s the need to make an assumption on the values of the pixels outside the image. There are different ways to handle the image edges, in this work the cropping method is accomplished, because with this technique any pixel in the output image which would require values from beyond the edge is skipped.

3. Parallelized Kernel Processing with C++11

As we can deduct from the description in chapter 1 and 2, kernel processing is an embarrassingly parallel algorithm. In fact, the convolution of the kernel with the image is calculated independently for each pixel of the image. As the dimensions of the kernel grow, the complexity of computing the convolution increases.. Since this task is used for each pixel of the image, the entire algorithm results very onerous. In the same way, as the image dimensions increase, the algorithm gets slower. Therefore, we decided to parallelize the convolution's phases, which are independent, in order to decrease the execution time.
For the parallelization we use the C++11. This version of the standard C++ provides both low and high level facilities for multithread programming.
Since the algorithm is embarrassingly parallel we don't need to handle race condition. In order to parallelize the convolution's phases on each of the image pixels, we have to choose the number of threads which perform the convolutions over the image pixels. Then we have to assign to each thread an image area where it will calculate the convolution for all the pixels of that area. Finally we have to be sure that all thread finish their execution, which can be done in C++11 using the command `join()`.

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.06s	0.04s	0.02s	0.02s	0.02s
1920 x 1200	0.63s	0,24s	0.11s	0.11s	0.011s
3939 x 2955	2.61s	1.21s	0.36s	0.33s	0.35s
8160 x 6120	9.66s	4.36s	1.30s	1.25s	1.23s

Table 1. Execution times of sequential and C++11 versions using a 3x3 kernel

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.19s	0.09s	0.045s	0.046s	0.041s
1920 x 1200	1.40s	0,62s	0.25s	0.21s	0.21s
3939 x 2955	6.44s	2.60s	0.79s	0.85s	0.83s
8160 x 6120	24.90s	10.46s	2.92s	3.05s	3.12s

Table 2. Execution times of sequential and C++11 versions using a 5x5 kernel

image dim.	seq.	chan.	block.	row	cols
508 x 493	0.46s	0.17s	0.08s	0.07s	0.07s
1920 x 1200	2.36s	1.08s	0.36s	0.35s	0.37s
3939 x 2955	11.45s	4.80s	1.50s	1.59s	1.44s
8160 x 6120	47.66s	19.25s	5.58s	5.76s	5.89s

Table 3. Execution times of sequential and C++11 versions using a 7x7 kernel

3.1. Experiments

We have conducted different experiments in order to test the performances of the parallelized implementation.
We have used images of different sizes which rang from 508×493 to 8160×6120. We have also used kernels of different dimension: 3×3, 5×5 and 7×7.
Moreover, we have assigned to threads different image areas: rows, columns and blocks. Also a parallelization on image channels has been done (using one thread per channel).
We have measured the execution time for both the sequential and the parallelized implementation of

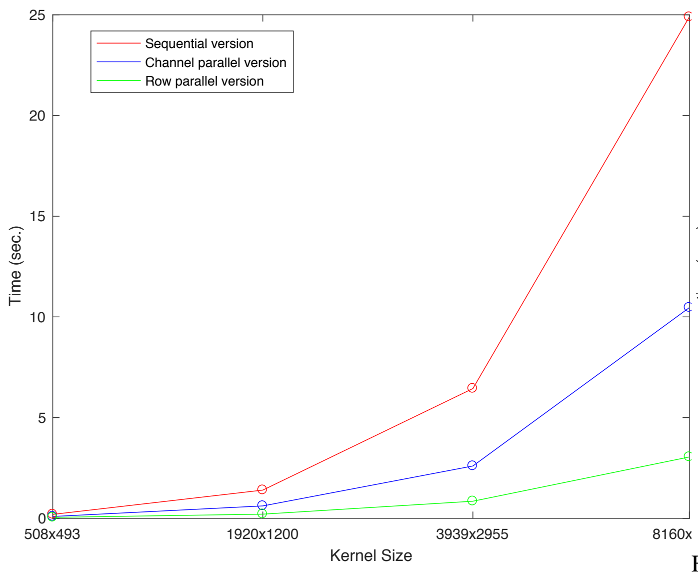

Figure 1. Execution time of sequential version and per rows and channels parallel versions.

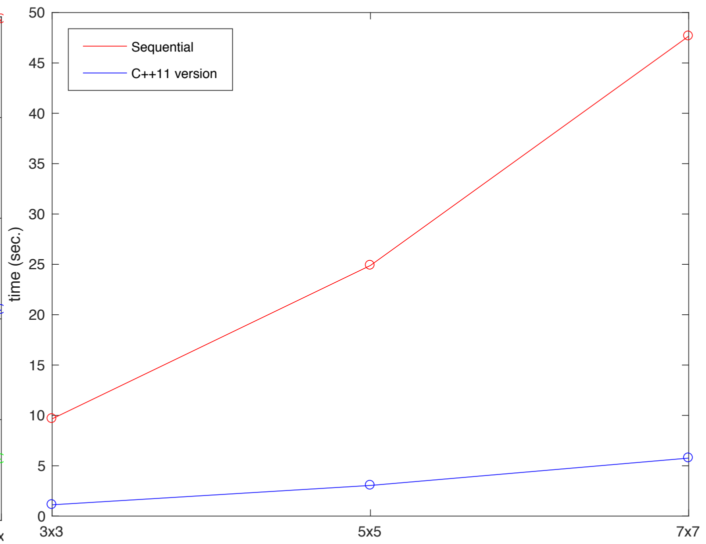

Figure 2. Execution time of sequential version and per rows parallel version using an image of size 8160x6120 and variable kernel dimensions.

the algorithm. The results are shown in the Table 1, Table 2 and Table 3. Each table refers to one of the three kernel dimensions taken into account. We notice that the execution time of both the sequential and the parallelized versions increases as the kernel size grows. As expected the execution time increases also as the image dimensions grow. For all the parallelized version (except channel version) 16 threads have been used.
Two very interesting analysis are the comparison between the different versions of parallelized algorithms and the evaluation of the speedup between the sequential and the parallelized versions. For what concerns the first, we notice that all the three versions (per row, per column and per blocks algorithms) have a similar execution time. For what regards the second, as we can see in Table 2 and in Figure 1 the speedup of the parallelized implementation on image channels is 2.38 using a kernel of 5×5 and 2.47 using a kernel 7×7; the speedup of other versions goes from 7.98 to 8.5 using a kernel of 5×5 and 8.1 to 8.5 using a kernel of 7×7. In Figure 2, we show the trend of the execution time for both the sequential and the parallel per rows versions in variation of the kernel size.

4. Parallelized Kernel Processing with CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. This platform is designed to work with C, C++,Fortran and Python. It enables explicit GPU memory management. The GPU runs multiple threads in parallel. A CUDA program is made of two parts: the host code and the device code; the host code runs on the CPU and the device code on the GPU.
In the implementation of Kernel Processing with CUDA we follow these steps:

1. We allocate GPU memories.
2. We copy image matrix, mask matrix and output matrix from CPU memory to GPU memory.
3. We execute the CUDA function to compute kernel convolution.
4. We copy data back from GPU to CPU.
5. We destroy GPU memories.

In the GPU, threads are organized in block and blocks constitute a grid. The structure of the device memory is relevant as well. We have the global memory that is the biggest on the device but it's slow (it is accessible for reading and writing by all threads of the grid), the shared memory that is fast but very small (it is accessible for reading and writing by all threads of the block), the constant memory ideal to store data that can be accessed uniformly and only for reading. These different kinds of memory bring to different approaches in the implementation of kernel convolution with CUDA. We realized four different implementations of kernel convolution: global memory, global memory and kernel in constant memory, shared memory and shared memory with kernel in constant memory.

4.1. Experiments

Also in this case we have done different experiments.
We have used images of different size, the same used for the experiments made for C++11. Kernel's size has been set to 3×3, 5×5 and 7×7.
In these experiments the aim is not only to show the difference between the sequential version and the the parallelized versions, but also to compare the various CUDA versions.
We have measured the execution time for the algorithm, varying the kernel size, for all the CUDA versions. As we can see in Table 1, Table 2, Table 3 , the CUDA versions are much faster than the sequential version. The results shown in the table don't take into account the allocation on GPU time. This time varies from $0.25msec$ to $0.40msec$ (second to the image which is used). Probably these very low execution times are also due to the fact that a GPU TITAN X has been used for the experiments.
In order to show the differences between the CUDA versions, another kind of experiment has been made. Each algorithm has been executed ten times over four images of different size. The execution time has been measured as the time of the entire process. We observe that the versions which used kernel in constant memory (global + const and shared + const) improved by little the execution time of the versions without constant memory. The fastest version,as expected, is the one with shared memory and kernel in constant memory. Anyway the difference in the execution time between the shared + const and the one with global memory (the slowest CUDA version) is very small.

image dim.	global	glob+const	shared	shared+const
508 x 493	0.027ms	0.026ms	0.024ms	0.023ms
1920 x 1200	0.027ms	0.027ms	0.025ms	0.024ms
3939 x 2955	0.027ms	0.027ms	0.025ms	0.025ms
8160 x 6120	0.029ms	0.028ms	0.027ms	0.026ms

Table 4. Execution times of CUDA versions using 3x3 kernel

image dim.	global	glob+const	shared	shared+const
508 x 493	0.026ms	0.026ms	0.025ms	0.025ms
1920 x 1200	0.028ms	0.027ms	0.025ms	0.025ms
3939 x 2955	0.029ms	0.028ms	0.027ms	0.026ms
8160 x 6120	0.030ms	0.028ms	0.027ms	0.026ms

Table 5. Execution times of CUDA versions using 5x5 kernel

image dim.	global	glob+const	shared	shared+const
508 x 493	0.029ms	0.029ms	0.026ms	0.025ms
1920 x 1200	0.030ms	0.030ms	0.029ms	0.028ms
3939 x 2955	0.031ms	0.031ms	0.030ms	0.030ms
8160 x 6120	0.031ms	0.030ms	0.029ms	0.028ms

Table 6. Execution times of CUDA versions using 7x7 kernel

5. Conclusions

In this paper we have presented the kernel processing algorithm and different parallelized implementations of it. We have analyzed the benefits in terms of execution time. We have observed that in all the experiments there are improvements brought by the parallelized versions. These improvements become relevant when the image size increases.
A search possibility for future works could be the one of implementing a different version of the algorithm using another technique to handle the edges of the image.