PC-2019/20 Kernel Image Processing

Matteo Amatori, 7024736
University of Florence
matteo.amatori@stud.unifi.it

Abstract

For the second project in the Parallel Computing Course I decided to implement a sequential and a multithreaded version of a Kernel Image Processing. Kernel Image Processing is a convolution process between an image and a kernel, that allows to apply different effect to a picture, which comprehend blurring, sharpening and edge detection. This project has been conducted by implementing two different versions of the algorithm, a sequential version that uses the CPU, implemented with the aid of the library OpenCV and a parallel version that uses the GPU, implemented with the aid of CUDA and the library LodePNG.

1. Introduction

This project aims to verify the theorical aspects of parallelisation. By adopting computing techniques that make use of more than one single thread for the computation of an algorithm, the time required by a program to run is inversely proportional to the number of processing units used. In addition to this, CUDA offers a hardware architecture that allows the developer to massively enhance the amount of processed data in a given amount of time. This is achieved by creating thousands of parallel threads running at the same time, each processing a different portion of the input data. CUDA is designed to handle extreme parallelizable data, like neural networks or image convolution.

Kernel Image Processing is based on image convolution and it is inherently extremely parallelizable. This project uses the Kernel Image Processing tecniques in order to apply a blur effect to a picture given as input.

2. Theorical analysis of the algorithm

In general, a process of convolution can be expressed by the following equation

$$g(x,y) = w \cdot f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t)$$

where $g(x,y)$ is the filtered image, $f(x,y)$ is the original image and ω is the filter kernel.

There are many different kernels that can be adopted in order to apply different effects to the images. In this project I decided to apply a box blur effect, which has a kernel matrix given by

$$\omega = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The algorithm proceeds to loop on every pixel of the image. For every specific pixel, the algorithm executes a multiplication value-by-value of the pixel with the corresponding kernel matrix entry, and then sums together all the values within a square whose dimensions are the same as the kernel matrix. In this specific implementation, every processed pixel's value is the result of the sum of all values obtained by multiplying the 9 surrounding pixel values (which comprehends the center pixel of the square) by the corresponding kernel matrix entry.

In order to prevent out-of-boundaries errors I added a padding composed of 0 values in the sequential version and a boundaries check for the

parallel one. This ensures that the computation of the corner and edge values is influenced only by the pixel values. For example, let's suppose that we consider the pixel $p_{i,j}$ where $i = 0$ and $j = 0$, which is the top-left corner of the image. In this case, the processed pixel's value $P_{i,j}$ is given by the following expression

$$P_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & p_{0,0} & p_{0,1} \\ 0 & p_{1,0} & p_{1,1} \end{bmatrix} \cdot \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In case of an edge processed pixel, for example $p_{i,j}$ where $i = 0$ and $j = 3$, the processed pixel's value $P_{i,j}$ is

$$P_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ p_{0,2} & p_{0,3} & p_{0,4} \\ p_{1,2} & p_{1,3} & p_{1,4} \end{bmatrix} \cdot \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In case of an edge processed pixel, for example $p_{i,j}$ where $i = 2$ and $j = 4$, the processed pixel's value $P_{i,j}$ is

$$P_{i,j} = \begin{bmatrix} p_{1,3} & p_{1,4} & p_{1,5} \\ p_{2,3} & p_{2,4} & p_{2,5} \\ p_{3,3} & p_{3,4} & p_{3,5} \end{bmatrix} \cdot \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The process computes every pixel in the image into new values $P_{i,j}$ and composes a new image with all the results of the computations.

2.1. Metrics used for the analysis

In order to evaluate the performance gain I took 10 different time measurement of 2 different input image sizes, both for the sequential and the parallel version of the algorithm. Such a number of repetitions for every possible combination of image size and technology used will avoid possible fluctuations in time measurements. Then, I took an average of the time measurements of all the categories. I proceeded to compare the sequential and the parallel version measurements in order to analyze the time elapsed difference between the two implementations, by using the Speedup metric

$$S_{latency} = \frac{T_S}{T_P} \tag{1}$$

where T_S and T_P represent the computation time of the sequential and the parallel algorithm.

3. Technologies used

In order to implement a Kernel Image Processing algorithm I choose to use C++17. This language is very versatile, as it offers Object-Oriented Programming support, while having low-level programming languages performance. For the parallel version of the algorithm I adopted CUDA as a parallelization tecnique.

In order to import and then output the images I used two different Computer Vision libraries. For the sequential version of the algorithm I used OpenCV[1], an open source library developed by Intel that provides an easy to use interface to import image files and distinctively read each color band values. For the parallel version I used LodePNG[2], a PNG encoder/decoder that provides an easy way to directly read PNG images into arrays.

The system that I used to run the benchmarks has a Ryzen 7 3700X, with 8 physical cores with virtualization, for a total of 16 computing units. It has a base clock of 3.6GHz and a boost clock up to 4.4GHz. Regarding the GPU, the system that I used to run the benchmarks has a NVIDIA GeForce GTX 1080, with 2560 CUDA cores and 8GB GDDR5X of local storage. This GPU has a base clock of 1607MHz and a boost clock up to 1733MHz.

4. Sequential algorithm implementation

My implementations of the sequential and parallel versions of this algorithm differ in some aspects. The first one is the library used to read and write the image. For the sequential version I adopted OpenCV. The process of reading and writing an image in C++ can be done with a few lines of code thanks to this library. I used the command `imread` in order to store the image inside a `Mat` object, a n-dimensional single or multi-channel array. In this specific implementation, I used a multi-channel array, since the images used for testing are RGB. After the reading has been done, I used a class called Image that contains 3 bi-dimensional vectors of integers, to store the values of each separate band R, G and B

and process them separately. The class has also 3 other bi-dimensional vectors of floats, that will contain the processed values of the RGB bands.

I then proceeded to add a padding of 0 values in order to prevent out of bound errors. This way I looped upon every value of each band, keeping an accumulator value that I incremented with the result of the multiplication of the considered band value by its corresponding kernel matrix entry. Then I assigned the accumulated value to the corresponding processed band value.

After this process, I managed to substitute the pixel values of the `Mat` object with the processed ones and output the image with `imwrite` . The image results blurred, as I expected.

5. Parallel algorithm implementation

The parallel version of the algorithm doesn't differ from the sequential one only in the processing aspects. In fact, I adopted a different library in order to read the image, since I wasn't sure that CUDA could handle vectors, and the choice was LodePNG. This library allowed me to store the pixel values of a PNG image of n total pixels into an array with the form of

$$img_{in} = (R_1, G_1, B_1, T_1, ..., R_n, G_n, B_n, T_n)$$

where R_i, G_i, B_i are the respective band value of the i-th pixel and T_i represents its transparency.

In order to properly process the image and apply a blur effect, I managed to isolate the 3 RGB bands. Hence the shape of the input is the following

$$inputImg = (R_1, G_1, B_1, ..., R_n, G_n, B_n)$$

For the computation itself, at first I created a filter function that accepts as input the 3 RGB bands and takes care of all the operations that are needed to copy onto the GPU memory the input image. This is done by allocating 2 arrays of size `width*height*3` and copying the input image to one of the 2 arrays just allocated. This ensures that, as the computation proceeds, the GPU can write the results on the second array.

Regarding the kernel, various different dimensions have been tested, but at the end I went with 1-dimensional block of 512 threads for each block and a 1-dimensional grid of

$$\frac{width * height * 3}{blockDim.x}$$

blocks. The result is a 2-dimensional kernel. This ensures that the process is run a number of times equal to the number of total pixels and it also makes it easy to identify each individual pixel inside the kernel function.

Each thread that is launched inside the grid has a unique identifier, which tells the thread what pixel to process. This information is given by the relation

$$pixel = blockIdx.x \cdot blockDim.x + threadIdx.x$$

Then the x index and the y index of the pixel are calculated using

$$x = pixel \bmod width \quad \text{and} \quad y = \frac{pixel - x}{width}$$

These two index values will become handy during the effective computation sections, since in this implementation I opted not to add a padding to the image like in the sequential version. In the parallel version, the biggest difference in performance is that the program never really loops on the whole image, pixel by pixel. Every thread that is launched on the GPU processes exactly one pixel by itself and stores the value on the output array at the correct location, completely indipendent from the other threads. This is guaranteed by the fact that the value of the processed pixel is completely not dependent by anything other than the input image pixels and the kernel matrix, and both never get modified during the whole computation.

The only loop that can be seen is the loop upon the kernel matrix to accumulate the values onto the output image vector. Note that, thanks to a check to protect from out-of-bounds situations, the accumulator gets updated only when the kernel matrix values are within the image boundaries.

Unit of measure [ms]	Sequential version (CPU)		Parallel version (GPU - CUDA)		
Measurements	512x512 Pixel	1920x1080 Pixel	512x512 Pixel	1920x1080 Pixel	3840x2160 Pixel
1	12.138	98.296	0.029	0.029	0.039
2	11.286	108.543	0.035	0.030	0.032
3	11.602	98.148	0.028	0.030	0.038
4	12.078	101.256	0.039	0.029	0.033
5	12.011	98.666	0.026	0.035	0.031
6	12.159	88.341	0.028	0.031	0.030
7	12.406	103.228	0.030	0.039	0.042
8	10.671	106.305	0.030	0.030	0.039
9	8.322	114.613	0.027	0.041	0.042
10	12.073	102.694	0.022	0.033	0.040
Total pixels	262144	2073600	262144	2073600	8294400
Mean	11.475	102.009	0.029	0.033	0.037
Std. dev.	1.221	7.065	0.005	0.042	0.046

Table 1. Kernel Image Processing performance analysis

At the end, each thread writes the accumulated values of the respective band in the proper position on the output array. When the kernel function computation ends, the output array that contains the processed pixels is copied back from the device to the host memory and the device memory is freed. At this point the output image is reconstructed, refilled with the transparency value and finally encoded to the file system.

6. Analysis of performance

The tests have been conducted with the aim of evaluating the performance gain derived from using a multi-threaded approach compared to a single-threaded one. The metric that is used to evaluate the gain is the Speedup, which is shown in formula (1). The code has been set to run, both in sequential and parallel mode, with 2 different images sizes: 512x512 pixels (for a total of 262144 pixels) and 1920x1080 pixels (for a total of 2073600 pixels). In addition to these two test sizes, I also decided to run ten measurements using a 4K image, which has a total of 8294400 pixels. Table 1 shows the results achieved. All the time measurements refer to nanoseconds.

Using the formula (1) we can easly take the mean values of the measurements taken and obtain the Speedup value.

First, the 512x512 pixels image has a Speedup value of

$$S_{latency} = \frac{T_S}{T_P} = \frac{11.475ms}{0.029ms} = 395.69$$

while the FullHD image shows a Speedup which is even higher than the first one

$$S_{latency} = \frac{T_S}{T_P} = \frac{102.009ms}{0.033ns} = 3091.18$$

The 4K image took on average 0.037ms, which is only 0.004ms more than the FullHD image. This allows me to think that even a 4K image wasn't a huge workload for the GPU.

7. Conclusions

With this project has been possible to prove the theorical aspects of the parallel programming paradigm and, in particular, to explore how powerful the CUDA technology can be when the given problem is easly parallelizable. Hence the performance gain, compared to just using a parallel version of the same algorithm on the CPU, can be orders of magnitude higher.

References

[1] "Opencv." `https://opencv.org/`.

[2] "Lodepng." `https://lodev.org/lodepng/`.