Scale Space Blob Detection: a parallel approach

Paolo Innocenti
paolo.innocenti@stud.unifi.it

Johan Andrey Bosso
johan.bosso@stud.unifi.it

Abstract

This paper describes the parallelization of a scale space blob detection algorithm, through the usage of the PyCuda library and the Python MultiProcessing library, before analyzing their performance differences and the improvements when compared to a sequential approach.

Future Distribution Permission

1. Introduction

The goal of this paper is to analyze the performance improvements obtained with the parallelization of a scale space blob detection algorithm, based upon the PyCuda[2] library and the Python MultiProcessing library, for scale space blob detection in digital images. We will first explain the problem of blob detection and the difference of gaussians algorithm used in order to obtain the scale space representation. Then, after describing the details of both the sequential implementation and the two parallel implementations, we will discuss the tests conducted in order to evaluate the performance differences between the three implementations and the improvements obtained with the usage of parallel programming techniques.

2. Blob Detection

In Computer Vision, blob detection aims at detecting regions, within digital images, that differ in properties, such as brightness or color, when compared with the surrounding regions. Usually, in these regions, these properties can be considered either constant or almost constant.[5]
In this context, the approach based on the difference of gaussians offers a good approximation of the traditional method based on the laplacian of gaussians, but does so by creating a scale space representation that enables the detection of blobs at different scales.

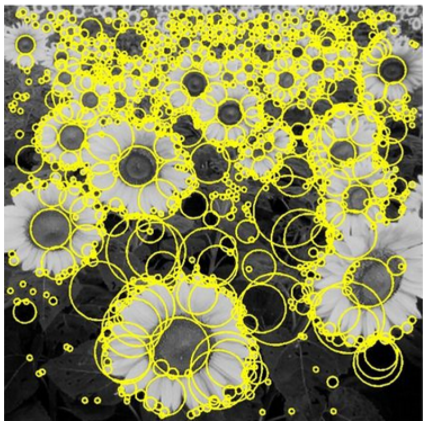
Figure 1. Blob detection example

2.1. Difference of Gaussians

Difference of Gaussians is a feature enhancement algorithm that involves the subtraction of one blurred version of an original image from another, less blurred version of the original, and is derived from an approximation of the laplacian of gaussians, making it most highly activated by a circle or blob. The subtraction of such images, obtained by convolving the original image with gaussian kernels obtained from different standard deviations, retains only the spatial information contained within the range of frequencies maintained by the blurred images. Consequently this method creates a band-pass filter that preserves only part of the information available in the orig-

inal image.
Moreover this technique uses a scale space representation, obtained by simply filtering the image with the use of kernels deriving from different sigmas, allowing the detection of blobs at different scales.
For an arbitrary image, we can summarize the process using algorithm 1.

Algorithm 1: Blob detection with difference of gaussians approach

```
Result: Blobs
1  generate pyramid of differences of gaussians
2  for $i \leftarrow 1$ to imageWidth-1 do
3      for $j \leftarrow 1$ to imageHeight-1 do
4          for $k \leftarrow 1$ to pyramidImagesAmount-1 do
5              candidate = dogPyramid[i, j, k]
6              cube = dogPyramid[i-1:i+1, j-1:j+1,
                 k-1:k+1]
7              if candidate == max(cube) or
                 candidate == min(cube) then
8                  save pixel coordinates
9              end
10         end
11     end
12 end
13 filter edges and non-extreme points from candidates
     and display result
```

In our approach, the first step has been to create a sequential implementation of the algorithm, which has been divided into three main sections:

1. Creation of difference of gaussians pyramid
2. Finding of extrema within the obtained pyramid
3. Candidates filtering and result display

In the first section, after loading the original image, we created the difference of gaussians pyramid by applying a gaussian blur based on different standard deviation and then subtracting the resulting images following the order defined at the beginning of paragraph 2.1.
In the third section, after detecting all of the possible candidates for blob centers, we first analyzed them in order to verify if they were edge points or if their contrast was below a certain threshold. Then we drew all the remaining candidates on the original images.
However, we put the main focus into parallelizing the second section of the program. In this section, we used the difference of gaussians pyramid created in the previous step by considering all the pixels of the pyramid's images, except for the pixels of the first and last images and those on the border of the other ones. We then considered these pixels as centers of 3x3x3-size blocks and we evaluated their eligibility to be a keypoint by verifying whether they were maximum/minimum within the block or not.

3. Implementation Details

We implemented the blob detector using Python[4], in its 3.7.5 version.
The first section was developed using the OpenCV[1] library in order to read the image and the Scikit-Image[3] library to generate the blurred images then subtracted in pairs to create the difference of gaussians pyramid.
The filtering in section 3 was instead obtained by discarding low contrast pixels and removing edge points accordingly to the SIFT[6] approach for the same problem. The result was then displayed by using the OpenCV library.
As for the second section, we first created a sequential version of the candidates-finding process by going through each pixel using a triple *for* loop, as shown in algorithm 1. Then we parallelized the section, first by using the PyCuda library and then by using the Python MultiProcessing one, in order to compare a GPU-based parallelization with a CPU-based one.

3.1. PyCuda

The first parallel implementation of the second section of the program was created using PyCuda 2019.1.2. The main idea behind this version has been to use 3x3x3-size Cuda blocks in order to evaluate the pixel in the center of the block, having $threadIdx.x = 1$, $threadIdx.y = 1$ and $threadIdx.z = 1$ within the block itself. Consequently, considering the K images of the dif-

ference of gaussians pyramid, each having size WxH, we set a size of (W-2, H-2, K-2) for the Cuda grid in order to compute all the candidates in one single iteration of the extrema-finding function.
Before passing the pyramid to the GPU, we applied a flatten to the 3-dimensional array in order to let Cuda work with a 1-dimensional array. Then we assigned the correct value of the array to each thread by pinpointing it with both the block and thread coordinates using the formula:

$$id = iS * (bZ + tZ) + iW * (bY + tY) + bX + tX$$

with id index of the pixel, iS amount of pixels for each image, iW width of the original image, bX, bY and bZ block's coordinates and tX, tY and tZ thread's coordinates within the block.
After assigning the correct value, we computed the maximum and the minimum within the block by using the *__shfl_down()* Cuda instruction, that allows the threads in the same warp to exchange data without the need for shared memory. This let us handle the reduction problem without having to first write the data to the shared memory and then synchronizing the threads before reading the data back from the memory.
Finally, after verifying that the maximum or minimum found was at the center of the block, each of the pixel coordinates(x=bX+1, y=bY+1 e z=bZ+1) was saved in an array used outside of the Cuda section.

3.2. MultiProcessing

The second parallel implementation has been based on the Python MultiProcessing library. This library, unlike the MultiThreading one, bypasses the CPython Global Interpreter Lock, enabling the use of multiple CPU cores available in modern architectures.
In our case, the library was used to remove one of the three *for* loops, used in the sequential approach, and parallelize the computation of the remaining double *for* loop. To achieve this, we first chose one of the three axes between image width, image height or pyramid depth that had to be removed from the loops. Then we created a MultiProcessing.Pool instance with one process for each value of the removed axis. Finally we computed the candidates for the remaining two *for* loops in each process.
In this implementation, we chose the pyramid depth axis instead of the two axes related to the image size because of the difference in their order of magnitude: the chosen one had a constant value of 17, caused by the constant standard deviation increase ratio used when generating the scale space, while the others had values ranging from 480 to 2160. Choosing one of the other two would have led to a longer execution time because of the frequent context switching within the CPU.

4. Performance Evaluation

4.1. Hardware Specifications

The testing has been performed mainly on a machine having CPU Intel(R) Core(TM) i5-6600K @ 3.6 GHz, 16GB DDR4 RAM, OS Ubuntu 19.10 and GPU EVGA(R) GTX950 SC+, with 768 Cuda Cores, 2GB DRAM and 1164 MHz Core Clock. However, in order to compare the performance of two GPUs belonging to different price ranges, tests of the PyCuda implementation were carried out also on another machine based on an NVIDIA(R) GTX980 with 2048 Cuda Cores, 4GB DRAM and 1405 MHz Core Clock.

4.2. Tests

For each of the implementations, we performed tests by changing the size of the used images in order to measure the execution time of the second section and compute the resulting speedup from the sequential approach. In particular, we considered three resolutions:

- 640x480
- FullHD
- 4K

We used 6 images for each resolution and then computed the average.
During the PyCuda implementation testing, we

also performed the profiling of the parallelized section to measure the Cuda code execution time on the GPU.

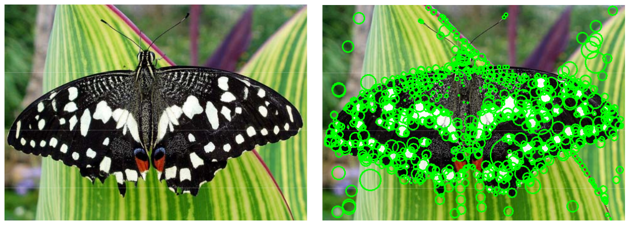
Figure 2. One of the test images before and after the blob detection

After completing the tests, we determined the speedup as

$$S = \frac{t_s}{t_p}$$

with t_s execution time of the sequential program and t_p execution time of the parallelized one.

5. Results

The results show a significant improvement of the execution time of the two parallel implementations in comparison with the sequential one. As

	Sequential		Processes		Pycuda	
	Avg(s)	σ(s)	Avg(s)	σ(s)	Avg(s)	σ(s)
640x480	23.5	3.1	8.1	0.2	0.24	0.01
FullHD	208.0	4.3	70.4	1.1	2.09	0.02
4K	696.7	11.6	211.5	5.2	6.9	0.2

Table 1. Test results

we can notice in table 1 and even better in figure 3, this difference in performance is significant for all the tested resolutions and it is especially remarkable when considering the PyCuda implementation. In fact, by using the computational power of the GPU, we reached a ~100x speedup even after considering the time needed to transfer the data to and from the GPU memory and to transform the results in order to have the same format as the one coming from the other approaches.

An interesting result emerged also from the profiling of the Cuda code. As shown in table 2, on the GTX950, and even more on the GTX980, the pure computation time of the extrema-finding function have been significantly lower than the CPU ones.
One final difference was noticed when comparing

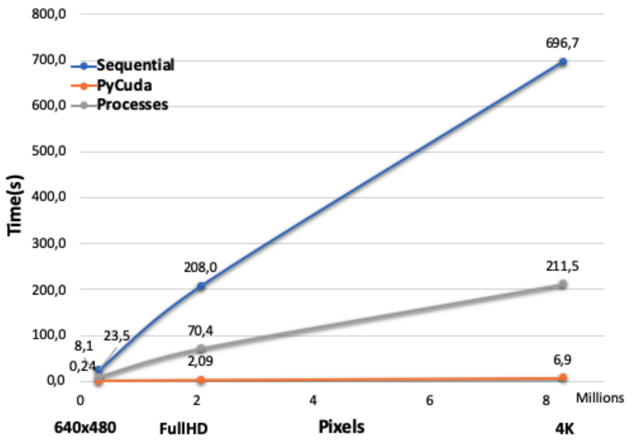

Figure 3. Execution time trends for different resolutions

the computation time of the two different GPUs as the 980 was able to compute the same function almost twice as fast as the 950. The profiling results are summed up in table 2 and figure 4.

	GTX980		GTX950	
	Avg(ms)	σ(ms)	Avg(ms)	σ(ms)
640x480	19.0	3.1	46.8	3.3
FullHD	191.6	7.5	405.7	12.9
4K	553.6	11.6	1113.6	22.3

Table 2. GTX980 and GTX950 computation times

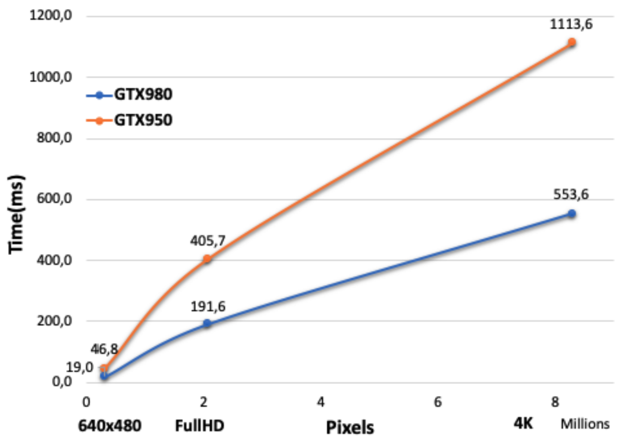

Figure 4. Cuda code execution time trends for GPUs

Finally, the obtained speedup is shown in table 3.

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

	Processes	PyCuda
640x480	2.90	97.83
FullHD	2.95	99.44
4K	3.29	100.75

Table 3. SpeedUp

[2] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[3] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[4] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[5] Wikipedia. Blob detection, 2019. [Online; Checked on 11/02/2020].

[6] Wikipedia. Scale-invariant feature transform, 2019. [Online; Checked on 11/02/2020].