PC-2024/2025 Performance of Blob detection in images using CUDA

Edoardo Branchi
E-mail address
edoardo.branchi@edu.unifi.it

Abstract

This report presents a performance analysis of a sequential and CUDA-based version of a blob detection algorithm using the difference of Gaussian (DoG) method. The sequential implementation is using CPU processing power, while the CUDA version uses GPU to parallelize the tasks. The performance of both versions are evaluated using a dataset of grayscale images. Experimental results demonstrate significant improvements in runtime efficiency with the CUDA implementation, particularly for large images and large number of pyramids.

1. Introduction

Blob detection is an algorithm that identifies region in an image that differ in properties like colors or brightness. These regions, are called "blobs,".The method I used is based on Difference of Gaussian (DoG).

1.1. Key characteristics

The Dog implementation of Blob detection is used in various different applications like object detection and feature extraction because of the advantages it brings but also has some flaws like:

- Advantages:
  - Simple and computationally efficient.
  - Works well for circular blob detection.
  - Scale-invariant (blobs detected at different scales).
- Disadvantages:
  - Sensitive to noise.
  - Does not perform well on non-circular or highly irregular blobs.

The motivations for the disadvantages are that using blurring a very noisy image can give unpredictable results. The blurring also is the cause why only circular shapes maintain the original shape, so different shapes can be altered easily making their detection more difficult.

1.2. General implementation

The key steps of the Blob detection algorithm are listed below:

- **Gaussian Pyramid Calculation:** The algorithm creates a Gaussian pyramid by applying Gaussian blur to the input image at multiple levels (Scales). The scale factor is controlled by a parameter k, which determines the progression of standard deviations (sigma) across scales.
- **DoG Pyramid Calculation:**Consecutive levels of the Gaussian pyramid are subtracted to compute the DoG pyramid. Each DoG image highlights zones of the image with a strong variation st a specific sale.
- **Local Min/Max Detection:** The algorithm identifies local minimum or maximum in a 3x3x3 zone across adjacent DoG images. A pixel is considered a min or max if it is either greater than or less than all its neighbors.
- **Keypoint generation:**For each point found in the previous step, a keypoint is created with its position and scale (sigma), which represent the blob's size in the image.

2. My Implementation Overview

My implementation focuses on keeping both versions as similar as possible to each other to obtain a significant performance measure. Both versions use a sequential version of Gaussian Blur provided by OpenCv. An attempt to parallelize it was made but the difference in the application of blurs made the two version diverge in the number of keypoints detected making the two version uncomparable given the fact that with different blurs the computation time could differ even with the same scale, sigma and image.

2.1. Sequential Blob Detection

The sequential code is a classic and simple example of Blob detection, it takes an image, convert it into a gray scale, and process it. There is also a commented out output function to save the image with the keypoints highlighted to compare the results.

2.2. Parallelized Blob Detection (CUDA)

The parallelized version is a bit more complicated. It take a photo and calculate the Gaussian blur in the CPU using openCV built-in function. There are two main functions that utilizes CUDA to compute data in a parallelized way:

- **findDifferenceKernel**:
  Each thread computes the difference between corresponding pixels in two Gaussian-blurred images using the global thread index computed using blockIdx, blockDim, and threadIdx. Each thread should compute a single pixel, in this way all the computation for a couple of gaussian blurred image is done at once. Also the memory is allocated using coalesced access patterns so consecutive threads access consecutive memory locations reducing waste.This should give us a significant increase in performance relative to the sequential implementation.
- **findExtremaKernel**:
  This function finds local maximum and minimum across 3 consecutive Dog images. It takes 3 images as an input, every thread process a single pixel and watches a 3x3 neighbor on 3 images, so a 3x3x3 check. if a pixel is greater of all it's neighbors its a maximum, if it is smaller it is a minimum. For the memory the technique is the same as the previous function.

The memory allocation is calculated to fit results coming from the functions and after the blurring the array is transferred to the device to continue the calculation. The blockSize will vary during the tests but the gridSize is calculated to cover the image using it's width and height. In various points of the implementation, for the sake of simplicity there are some boundary checks or skip to avoid to handle border pixels.

3. Experimental Setup

The experiment are run on a remote machine called "papavero" equipped with:

- **CPU:** 2x Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz for a total of 64 cores.
- **GPU:** NVIDIA RTX™ A2000 12GB version.
- **RAM:** 64 GB

The execution it's handled by the build,execution and development of Clion configured to target the remote machine.

The tests will be conducted on 3 image sizes:

- **Small** image: 620x320 pixels.
- **Medium** image: 1280x853 pixels.
- **High** image: 2600x1498 pixels.

An example can be seen in Figure 1.

Figure 1. One of the images used for testing

On all dimensions we will test the execution time increasing the number of scales (Levels of the dog pyramid). Also we will try to use 16x16, 16x32 and 32x8 block sizes to see how it affect performances.

Given the fact that both implementation share the same blurring technique the execution time and speedup will be analyzed only considering the time used to compute the DoG and to find the extrema.

4. Results and Discussion

Using the same Blurring technique we achieved almost the same result in blob detection, not a trivial result given the very different way each version calculate results.

A sample output can be seen in Figure 2 and Figure 3.

The choice of a landscape image is to push the limit of blob detection, because they have a lot of dark and light areas and the main subject is always small in relation to the size of the images, so when blurring the line between extracting some important features and completely missing the object is very thin.

In both of this figure we can see some clusters of point along the hills in the back that separates them from the sky, also cloud are pretty well detected along with the tree in the middle and the hill in the foreground. This is a good indication that both implementation are working.

To achieve a more accurate results we can fine tune the parameters of both implementation to detect well object in

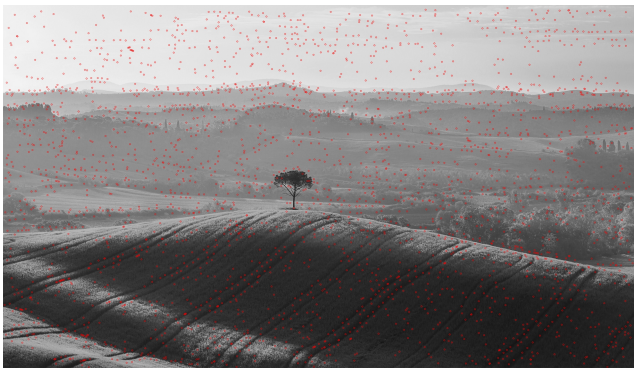

Figure 2. Output of the sequential implementation

this kind of wide landscapes images but this go out the scope of this report.

In the next section will be discussing about performance using parameters that not always produce good results in terms of detection but are a good indication about the overall performances of both implementation.

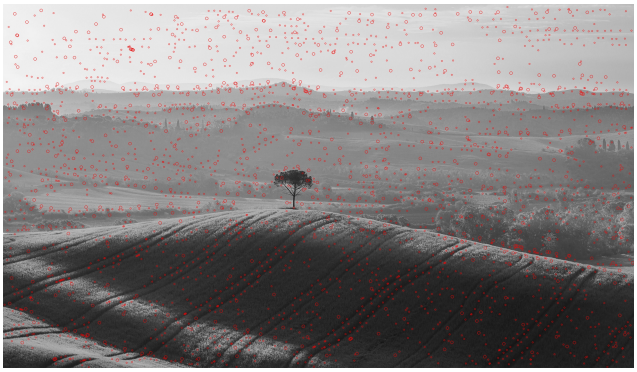

Figure 3. Output of the CUDA implementation

4.1. Performance Comparison

The following plots compares a typical 16x16 blocksize against the sequential implementation comparing the executions time only for the DoG computation and detecting of the extrema across all three image sizes. All measurement are an average of 5 different tests.

We can see that for small sized images the two execution time are very similar, indicating that the transferring of data between host and device memory overcome the computation time, a cuda implementation for this case won't be useful at all given the complication of the parallel implementation.

Things begin to differ for medium and large image size, where the cuda implementation outperforms the sequential one, this gap increases as the image gets bigger, so we can conclude that with an adequate computation load the parallel version is way faster at executing.

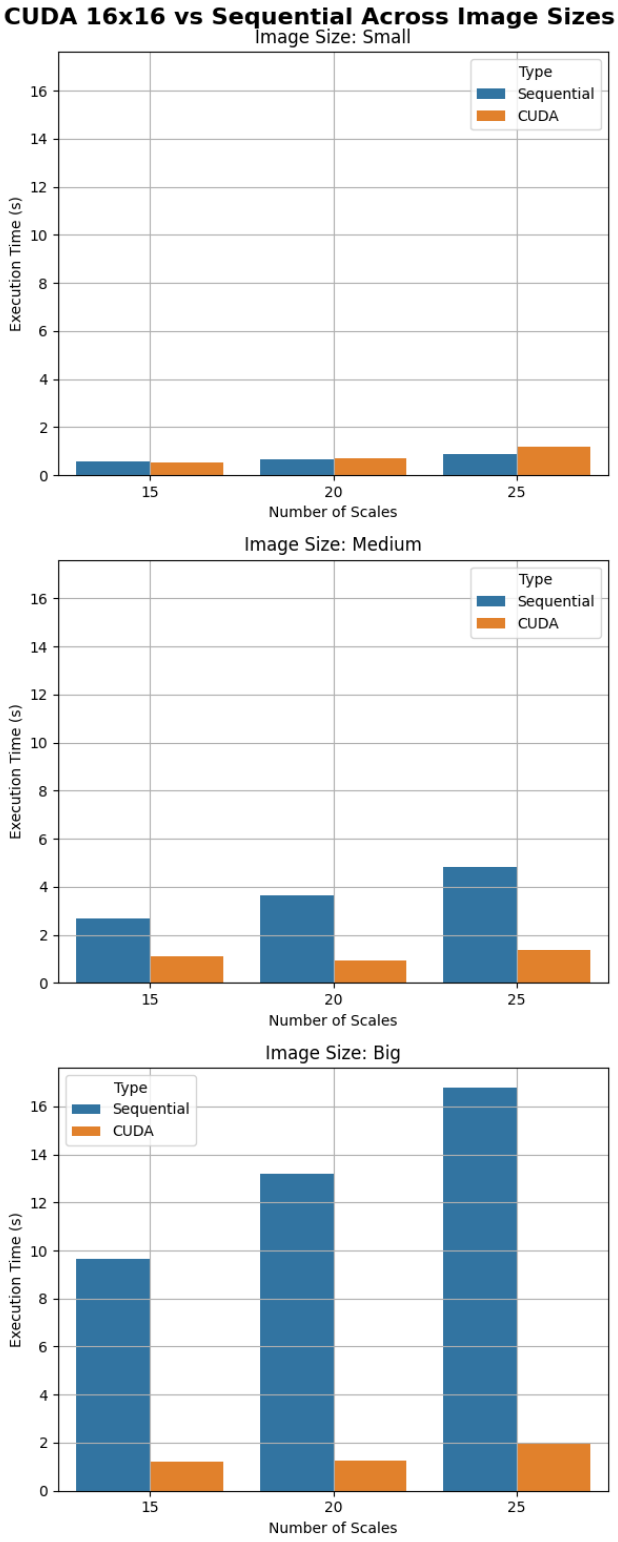

Figure 4. Cuda vs Sequential

Cuda offers the possibility of choosing the size of the blocks when executing, so the plot in 4 shows the execution times for different block sizes and images, the small image was omitted because, as said earlier, it would be pointless to compare the small images with other sizes as the sequential version is equivalent.

We can see that the 16x16 configuration is not performing as well as the 32x8, especially with a large number of scales, where the 32x8 is clearly the most performative one out of the three for heavy and small load.

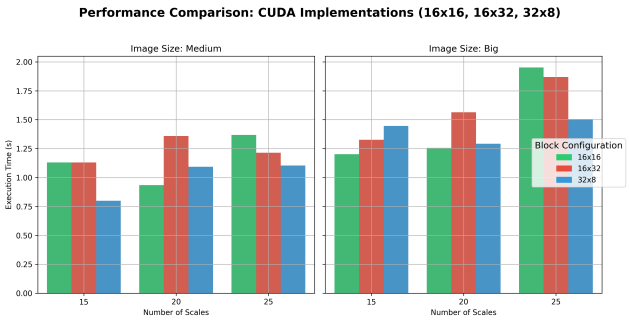

Figure 5. Cuda blockSize variation

The plot in figure 6 shows the overall speedup of various block configuration relative to the sequential one, for big images and large scales the 32x8 is the most performative of the three with a speedup of over 12, pattern that repeats also on the medium size images. The 16x16 block size start to lose utility when moving from 15 or 20 scales to 25 as wee can see it's the winner for 15 and 20 but lose traction when transitioning to 25 scales.

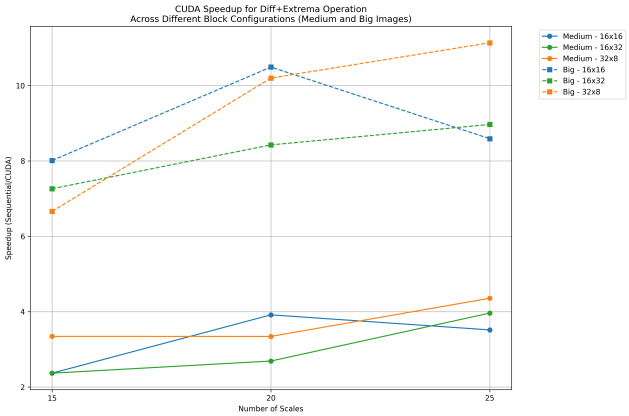

Figure 6. Speedup across multiple CUDA configuration against sequential

5. Conclusion

The CUDA code clearly outperforms the sequential implementation but brings with it a whole new level of difficulties writing code as well debugging it, while a 12x speedup could be an advantage it requires apart from specific GPU hardware a lot of conventions and rules to be functional. I would recommend using it only for heavy and sustained loads because the time needed to write the code for it could make the 12 speedup useless if the time used to write the program is enormously bigger.

6. GitHub repository

BlobDetection Sequential
BlobDetection CUDA