GPGPU : CUDA simple object detector

Mathieu Rivier
mathieu.rivier@epita.fr

Moustapha Diop
moustapha.diop@epita.fr

Othman Elbaz
othman.elbaz@epita.fr

Lucas Pinot
lucas.pinot@epita.fr

November 2022

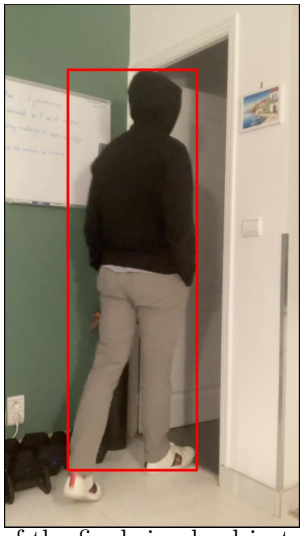

Figure 1: Output of the final simple object detection program

Abstract

This paper aims to bring a comprehensive understanding of simple object detection and the different steps that have been explored and implemented as part of this paper. This paper will explain the choices that have been made and showcase and compare the performance of each of the the three implementations (CPU, naive GPU, GPU) that have been written. As a way to accurately test performance, a benchmark that evaluates each stage of the object detection individually and then evaluates the program as a whole has been written for each of the three implementations.

Contents

1 Introduction to Object Detection

The aim of this project is to implement a simple object detector that can process videos and output a video with a red bounding box around the object(s). This paper is organised in 10 parts. Each of the first 6 are divided into a presentation of the global implementation and benchmarks and followed for some by the difficulties and optimisations implemented.

2 Grayscale

Grayscale is used in object detection because it is a very efficient way to store image data. When an image is stored in grayscale, each pixel is represented by a single number, instead of three numbers (red, green, and blue). This makes the image file smaller, which reduces the time it takes to download and open the image as well as performing other transformations on it.

Figure 2: Gaussian Blurred Grayscale image outputted from our program.

Grayscale is also used because it makes it easier to detect edges in an image. Edges are important features that can be used to identify objects in an image, edges detection is also mentioned in sections 3 & 5. When an image is in grayscale, the edges stand out more clearly than they do in a coloured image.

The representation of grayscale is simple. Each pixel in an image is given a value between 0 and 255, with 0 being black and 255 being white. The brightness of each pixel is determined by its distance from black. A grayscale image looks like figure 2.

2.1 Grayscale Benchmarks

Implementation	**Frame-rate (in frames/second)**
CPU	198.2
naive GPU	4.14K
GPU	/

Table 1: Grayscale Benchmarks

3 Gaussian Blur

Gaussian blur is used to reduce image noise and detail. It is used to smooth an image, otherwise known as a blurring the image which can be seen in figure 2. Smoothing an image can help reduce noise and make edges more pronounced, which can make it easier to detect objects in the image as it helps the thresholding see section 5.

Gaussian blur is implemented by convolving an image with a Gaussian kernel, which is a matrix of weights that is generated by a Gaussian function. The size of the kernel and the standard deviation of the Gaussian function determine the amount of blur that is applied to the image. The kernel is a square matrix with an odd number of rows and columns, where the value of each element is a weighted sum of the input pixels.

3.1 Gaussian Blur Implementation Feedback

Implementing gaussian blur algorithm is pretty straight forward. It has although required multiple optimisation passes both in CPU and GPU to yield reasonable performance.

3.2 Gaussian Blur Benchmarks

Implementation	**Frame-rate (in frames/second)**
CPU	15.23
naive GPU	359.35
GPU	/

Table 2: Gaussian Blur Benchmarks

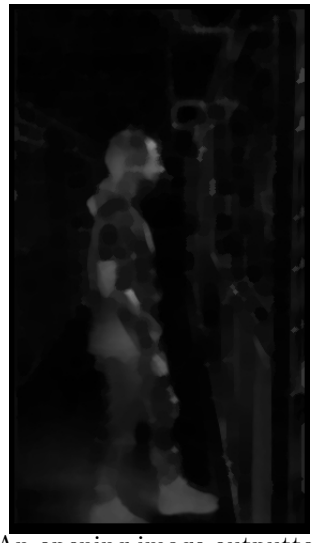

Figure 3: An opening image outputted from our program.

4 Morphological Closing & Opening

Morphological closing and opening are used in object detection because they can help to improve the contrast in an image. This can make it easier to detect objects in the image. Morphological closing can also help to fill in small holes in objects, which can make them easier to detect. Morphological opening can help to remove small objects from an image, which can also make it easier to detect the objects that are remaining in the image.

A morphological closing is an operation that is typically used to close small holes or gaps in objects. This is done by using a structuring element that is smaller than the actual object. The structuring element is then moved over the image and the pixels in the object that are within the structuring element are set to the maximum value of its neighbourhood (see section 6 about connectedness). The structuring element is then moved over the object again and the pixels that are within the structuring element are set to the minimum value of its neighbourhood. This process is repeated until all of the holes in the object have been closed.

A morphological opening is the opposite of a morphological closing. It is used to remove small objects from an image (see figure 3). This is done by using a structuring element that is smaller than the actual object. The structuring element is then moved over the image and the pixels in the object that are within the structuring element are set to the minimum value of its neighbourhood. The structuring element is then moved over the object again and the pixels that are within the structuring element are set to the maximum value of its neighbourhood. This process is repeated until all of the objects in the image have been removed.

4.1 Morphological closing & opening Benchmarks

Implementation	Frame-rate (in frames/second)
Closing CPU	10.38
Opening CPU	2.15
Closing naive GPU	1.20K
Opening naive GPU	328.00
Closing GPU	/
Opening GPU	/

Table 3: Morphological Closing & Opening Benchmarks

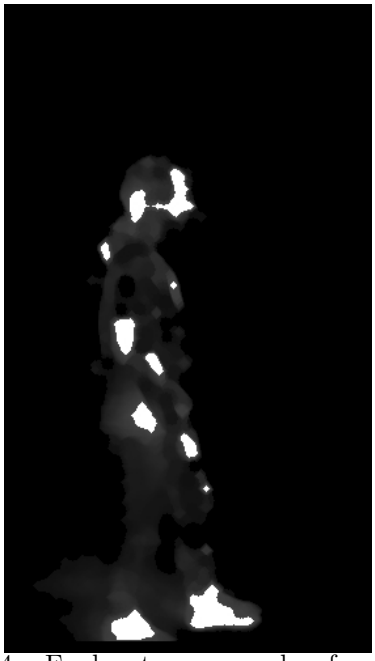

Figure 4: Explanatory example of connected components with two thresholds outputted from our program. Gray parts represent Otsu threshold, White parts the second threshold and black parts the background (In real program all highlighted parts are white)

5 Threshold - (Otsu & second threshold)

A threshold is a value that is used to determine whether a pixel should be considered part of an object or not. The threshold is typically set to a value that is between the pixel values of the object and the background. If the pixel value is greater than the threshold, then the pixel is considered part of the object. If the pixel value is less than the threshold, then the pixel is considered part of the background.

Thresholds are used in object detection because they allow for the removal of background noise that can interfere with the detection of objects. The threshold is also used to improve the accuracy of the object detection by ensuring that only pixels that are part of the object are considered.

The threshold is typically implemented as a function that takes in a pixel value between 0 and 255 as seen in section 2 and outputs a binary value (0 or 1) depending on whether the pixel should be considered part of the object or not.

We chose Otsu as our main threshold. This method is used in object detection because it can automatically find the threshold that best separates the foreground from the background, dividing the image in two parts as a normal threshold. Otsu thresholding is especially useful in images that have a lot of noise or background clutter (see figure 4). The method is named after Nobuyuki Otsu, who first described it in a 1979 paper.

Otsu thresholding works by looking at the histogram of an image and finding a threshold value that can best separate the image into two parts. The threshold value is chosen so that the sum of the variances of the two parts is minimised. This method is typically used for images that have two clearly defined parts, such as foreground and background.

We further use two thresholds. There are two main reasons for using two thresholds when computing connected components. The first is that it can help to reduce the amount of noise in the image. The second reason is that it can help to improve the accuracy of the segmentation.

Figure 5: An image with 2nd threshold based on otsu-threshold outputted from our program.

Using two thresholds can help to reduce the amount of noise in the image because it allows for more flexibility in the definition of what is considered to be a connected component. For example, consider an image with a lot of small, isolated points of noise. If we were to use a single threshold, then all of these points would be considered to be part of the same connected component (see section 6 for more detail on connected components). However, if we use two thresholds, then we can define a smaller threshold that excludes these points of noise.

Using two thresholds can also help to improve the accuracy of the segmentation. This is because it allows us to more accurately define the boundaries of the connected components. For example, consider an image with two connected components that are very close to each other. If we were to use a single threshold, then it would be difficult to accurately segment the two components. However, if we use two thresholds, then we can define a smaller threshold that more accurately defines the boundary between the two components.

5.1 Thresholds Implementation Feedback

Similarly to section 4, implementing the thresholds required more research and implementation choices. In fact, the first step was to find an algorithm that is capable of adapting the threshold to the current image/set of images it is currently processing. It was thus necessary to research and find Otsu. Secondly it was also a choice to add a second threshold to gain in precision and accuracy, and decide what to define the threshold as.

Once the general decision made, and implemented in CPU, the task of implementing those in GPU presented itself as the next challenge.

In order to detect bottlenecks in the threshold and connected components in the GPU implementations, a separate benchmark had to be created for each one. `BM_firsts_threshold_gpu` for thresholds benchmarks and `BM_connexe_components_gpu` for connected components.

5.2 Threshold Bottlenecks & Resolution

In order to compute an Otsu threshold it is required to separately compute the variance of the pixel values above and below a certain threshold value. In the naive GPU implementation, the mean, variance and number of pixels above and below a threshold were calculated separately using global memory.

In order to improve the threshold implementation the computation of the variance is now done in a single kernel computing both the variance and mean at once. In fact, the variance is mathematically obtainable without computing the mean beforehand. Furthermore, output pri-

vatisation was used in order not tap in global memory directly

5.3 Thresholds Benchmarks

Our implementations of connected components and the two thresholds entails that both of those parts are tested as a single unit. The following benchmarks thus represent the simultaneous results of both Threshold implementations and the connected components implementation. These two GPU optimisations enable 47 frames per second against 1.7 for the naive GPU implementation (see figure 11, 12, 15 for benchmarks screenshots).

Implementation	**Frame-rate (in frames/second)**
CPU	0.45
naive GPU	1.32
GPU	7.58

Table 4: Thresholds & Connected Components Benchmarks

6 Connected Components

A connected component is a maximal set of pixels in an image that are all connected to each other. Connectedness is defined using a connectivity criterion, which specifies how to determine whether two pixels are connected. The most common connectivity criterion is 4-connectivity, which says that two pixels are connected if they are adjacent to each other horizontally or vertically. The second most common connectivity criterion is 8-connectivity, which says that two pixels are connected if they are adjacent to each other horizontally, vertically or diagonally. For this paper we implemented 4-connectivity.

Connected components are used in object detection because they can be used to identify individual objects in an image. For example, if we have an image of a bunch of flowers, we can use connected components to identify each individual flower. To do this, we first threshold the image to create a binary image. This means that each pixel is either black or white, with black pixels representing background pixels and white pixels representing foreground pixels (see figure 6). Then, we compute the connected components of the binary image. This will give us a set of connected components, each of which represents an individual flower.

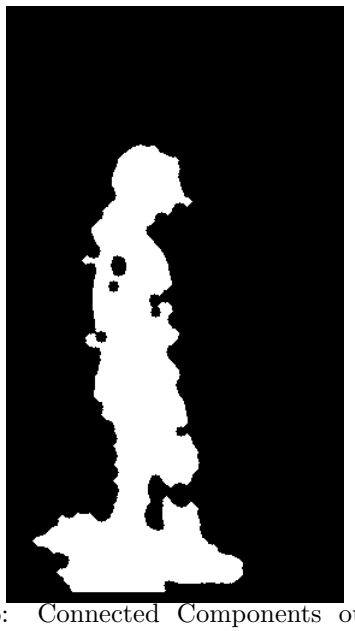

Figure 6: Connected Components outputted from our program.

Although, this can be improved by using a connected component algorithm with two thresholds as demonstrated in figure 4. A connected component with two thresholds is a process used in object detection whereby an image is first thresholded using a high threshold value (see figure 5) and then a low threshold value. The high threshold value is used to segment the image into foreground and background regions, while the low threshold value is used to identify the connected components within the foreground region. Said otherwise, the first threshold is used to find the edges of objects in the image, while the second threshold is used to find the boundaries of objects in the image. This approach is used because it allows for more accurate detec-

tion of objects, as well as for the identification of smaller objects that may be missed using a single threshold value. Finally, using two thresholds can help to improve the speed of object detection. This is because it can help to reduce the search space.

6.1 Connected Components Bottlenecks & Resolution

In the naive GPU implementation a convolution is applied on each pixel in order to propagate the brightest pixels information and get the connected components. In the optimised GPU implementation, tiling and memory privatisation in shared memory are used. The image's different blocks are stored on the shared memory and is subsequently used in a convolution using shared memory.

This optimisation enables the optimised GPU implementation to run at around 18 frames per second against 6 frames per second for the naive GPU implementation (see figure 11, 12, 15 for benchmarks screenshots).

6.2 Connected Components Benchmarks

see table 5 for the benchmarks.

7 bounding boxes

A bounding box is a rectangle that can be drawn around an object in an image. It is typically used to denote the area of an object that a machine learning algorithm should focus on when trying to detect that object. The bounding box can be implemented in object detection by first identifying the coordinates of the top-left and bottom-right corners of the rectangle. These coordinates can be determined by finding the minimum and maximum x and y values of the object's pixels. Once the coordinates are known, the bounding box can be drawn around the object by simply connecting the top-left and bottom-right corners with lines.

In this particular case, the coordinates for the bounding box are given by our bbox program written in C++/CUDA and then drawn in python.

8 Program benchmarks

Starting from a 0.34 Frame per second rate for the cpu implementation to a 17 frame per second rate in the optimised GPU implementation passing by 1.38 per second rate for the naive GPU implementation. This project has taught us a lot about image detection as well as working with gpus.

Implementation	**Frame-rate (in frames/second)**
CPU	0.34
naive GPU	1.38
GPU	17.29

Table 5: Final program Benchmarks

9 Conclusion

In this paper the authors managed to successfully implement a simple object detection program. The CPU and naive GPU implementations suffered from poor performance. Although, with the final performance implementation yields much better performance, and obtain 17 frames/seconds which represents a 42x improvement over the initial CPU implementation.

10 Image results

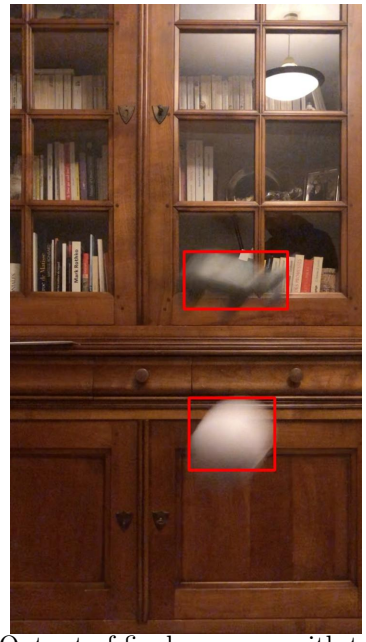

Figure 7: Output of final program with two objects.

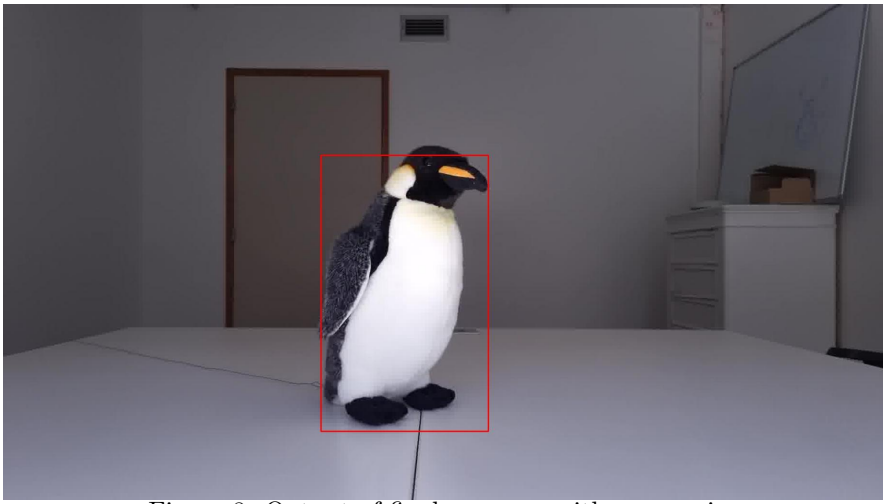

Figure 8: Output of final program with a penguin.

Figure 9: Output of final program of ER.

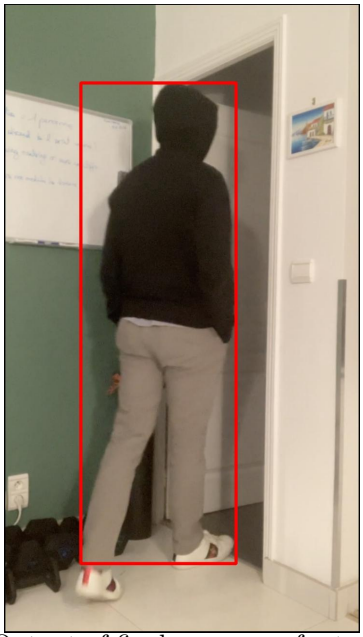

Figure 10: Output of final program of a team member.

11 Task distribution

	CPU	Naive GPU	Optimised GPU
Gray scale	Lucas / Othman	Lucas	
Blurring	Lucas / Moustapha / Mathieu	Othman / Lucas	
Difference	Moustapha / Mathieu	Othman / Moustapha	
Opening/Closing	Othman	Othman	Othman
Threshold	Mathieu / Moustapha	Moustapha / Mathieu	Moustapha
Connected components	Mathieu / Moustapha	Mathieu / Moustapha	Moustapha
Main	Moustapha / Mathieu	Moustapha	Moustapha
Benchmark	Moustapha	Moustapha	Moustapha
Report	Mathieu	Mathieu	Mathieu
Slides	Mathieu	Mathieu	Mathieu
Refacto	Mathieu / Moustapha	Moustapha / Mathieu	Moustapha / Mathieu
Vizualisation tool	Othman		

Table 6: Task distribution

12 Annex

```
-------------------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------
BM_gray_scale_cpu/real_time        3.07 ms         3.07 ms          180 frame_rate=325.765/s
BM_blurring_cpu/real_time          65.9 ms         65.9 ms           11 frame_rate=15.1663/s
BM_difference_cpu/real_time        6.23 ms         6.23 ms          181 frame_rate=160.409/s
BM_closing_cpu/real_time           95.0 ms         95.0 ms            8 frame_rate=10.5311/s
BM_opening_cpu/real_time            464 ms          464 ms            2 frame_rate=2.15738/s
BM_threshold_cpu/real_time         2151 ms         2151 ms            1 frame_rate=0.464851/s
BM_main_cpu/real_time              2908 ms         2908 ms            1 frame_rate=0.343841/s
```

Figure 11: CPU implementation Program Benchmark

```
-------------------------------------------------------------------------------------------
Benchmark                                   Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------------------
BM_gray_scale_gpu/real_time              0.240 ms        0.240 ms         2485 frame_rate=4.16146k/s
BM_blurring_gpu/real_time                 2.76 ms         2.76 ms          254 frame_rate=362.136/s
BM_difference_gpu/real_time              0.079 ms        0.079 ms         9026 frame_rate=12.6671k/s
BM_closing_gpu/real_time                 0.824 ms        0.824 ms          829 frame_rate=1.21286k/s
BM_opening_gpu/real_time                 3.01 ms         3.01 ms          233 frame_rate=332.585/s
BM_first_threshold_gpu/real_time          579 ms          579 ms            1 frame_rate=1.72801/s
BM_connexe_components_gpu/real_time       165 ms          165 ms            4 frame_rate=6.06015/s
BM_threshold_gpu/real_time                733 ms          716 ms            1 frame_rate=1.36469/s
BM_main_gpu/real_time                     720 ms          720 ms            1 frame_rate=1.38878/s
```

Figure 12: naive GPU implementation Program Benchmark

```
-------------------------------------------------------------------------------------------
Benchmark                                   Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------------------
BM_gray_scale_gpu/real_time              0.242 ms        0.242 ms         2471 frame_rate=4.1363k/s
BM_blurring_gpu/real_time                 2.78 ms         2.78 ms          251 frame_rate=359.378/s
BM_difference_gpu/real_time              0.078 ms        0.078 ms         8958 frame_rate=12.8077k/s
BM_closing_gpu/real_time                 0.829 ms        0.829 ms          845 frame_rate=1.20633k/s
BM_opening_gpu/real_time                 3.05 ms         3.05 ms          230 frame_rate=327.79/s
BM_first_threshold_gpu/real_time         20.6 ms         20.6 ms           35 frame_rate=48.5675/s
BM_connexe_components_gpu/real_time      55.6 ms         55.6 ms           12 frame_rate=17.9948/s
BM_threshold_gpu/real_time                132 ms          132 ms            9 frame_rate=7.58189/s
BM_bbox_gpu/real_time                    1.03 ms         1.03 ms          666 frame_rate=971.16/s
BM_main_gpu/real_time                    57.8 ms         57.8 ms           12 frame_rate=17.2918/s
```

Figure 13: GPU implementation Program Benchmark

Benchmark	Time	CPU	Iterations	UserCounters...
BM_gray_scale_gpu/real_time	0.240 ms	0.240 ms	2374	frame_rate=4.15957k/s
BM_blurring_gpu/real_time	2.76 ms	2.76 ms	253	frame_rate=361.85/s
BM_difference_gpu/real_time	0.078 ms	0.078 ms	8996	frame_rate=12.8574k/s
BM_closing_gpu/real_time	0.827 ms	0.827 ms	851	frame_rate=1.20977k/s
BM_opening_gpu/real_time	3.03 ms	3.03 ms	231	frame_rate=329.914/s
BM_first_threshold_gpu/real_time	20.7 ms	20.7 ms	34	frame_rate=48.2425/s
BM_connexe_components_gpu/real_time	55.6 ms	55.6 ms	12	frame_rate=17.9965/s
BM_threshold_gpu/real_time	101 ms	101 ms	9	frame_rate=9.89815/s
BM_bbox_gpu/real_time	1.06 ms	1.06 ms	659	frame_rate=941.013/s
BM_main_gpu/real_time	111 ms	111 ms	6	frame_rate=9.02866/s

Figure 14: GPU implementation Program Benchmark. Kernel size: opening(21), closing(41)

Benchmark	Time	CPU	Iterations	UserCounters...
BM_gray_scale_gpu/real_time	0.240 ms	0.240 ms	2541	frame_rate=4.16129k/s
BM_blurring_gpu/real_time	2.76 ms	2.76 ms	253	frame_rate=362.156/s
BM_difference_gpu/real_time	0.078 ms	0.078 ms	9018	frame_rate=12.7818k/s
BM_closing_gpu/real_time	0.616 ms	0.615 ms	1141	frame_rate=1.62329k/s
BM_opening_gpu/real_time	2.05 ms	2.05 ms	342	frame_rate=487.464/s
BM_first_threshold_gpu/real_time	21.0 ms	21.0 ms	35	frame_rate=47.6636/s
BM_connexe_components_gpu/real_time	55.7 ms	55.7 ms	13	frame_rate=17.9504/s
BM_threshold_gpu/real_time	101 ms	101 ms	9	frame_rate=9.93528/s
BM_bbox_gpu/real_time	1.06 ms	1.06 ms	658	frame_rate=939.722/s
BM_main_gpu/real_time	87.5 ms	87.5 ms	8	frame_rate=11.4286/s

Figure 15: Optimised Morpho GPU implementation Program Benchmark. Kernel size: opening(21), closing(41)