GPGPU
Project report

Alexandre Lemonnier - Sarah Gutierez
Victor Simonin - Adrien Barens

Promotion 2023

Contents

1 Description of the problem

The goal of this project is to implement a simple object detector in CUDA. There are four main objectives which are :

- Apply data-parallelism concepts
- Practice with CUDA
- Set up a benchmark with a sound evaluation procedure
- Present our results in a clear and convincing way

The algorithm steps are the following: first, it converts the image in grayscale, then it smooths the background image and the new image (we used a Gaussian filter) and compute the difference between the reference image and the input images. It performs a morphological closing/opening with a disk (or a rectangle) to remove non-meaningful objects. Finally, it thresholds the image and keep only the connected components with high peaks, and output the bounding boxes.

We decided to start early with a simple python implementation to get into the subject and to get the first results as fast as possible. Two C++ CPU implementations then followed, one with the *openCV* library and one without any helpers.

Finally, we tried to get a working C++/Cuda GPU implementation, get some strong benchmarks and iterate over it to detect our bottlenecks and improve over them.

2 Baseline of CPU implementation

First, a python version has been implemented with *openCV* to get our first results and to compare them later to the C++ CPU and C++/Cuda GPU implementations. The code respects here scrupulously the steps of the algorithm.

Then, the C++ CPU version was our main focus, the objectives were to reproduce the results of the previous version. We implemented a simple *openCV* version to match the python results and we implemented another version without any helpers.

For this last one, the grayscale, gaussian blur, difference, and closing/opening were quite straightforward without any parallelism. We first used a square as a structuring element for the closing/opening as it was easier. The most challenging part, at least to understand, was the threshold and the labeling of connected components. Here we decided to follow the indications in the subject and apply a first threshold on the resulting image of the closing/opening section, then compute the connected components, and finally, apply another threshold to keep only the relevant connected components.

3 Baseline of GPU implementation

The GPU implementation is composed of multiple kernels that computes quickly the steps of the algorithm. After each we choose to save the images to better understand the multiple operations on the images and check if everything work fine. Basically we transpose the CPU algorithms to GPU to accelerate the rendering.

A Grayscale Kernel

For this step, we had to convert a buffer with three channels into a single one. Basically, we apply the following function to compute the grayscale of each pixel: $Y_{linear} = 0.2126R_{linear} + 0.7152G_{linear} + 0.0722B_{linear}$ where R, G, and B represent the value inside the three channels.

A Gaussian Blur Kernel

To apply this filter we had to compute the gaussian kernel before applying it to each pixel. We use the following formula to compute the kernel: $G(x) = \frac{\exp \frac{-x^2}{2\sigma^2}}{\sqrt{2\pi\sigma^2}}$. After that we apply this kernel with every pixel we chose an arbitrary sigma to have a good smooth factor that can be easily observed.

A Difference Kernel

During this phase, we take the reference image, grayscaled with gaussian blur, and an input image also grayscaled with gaussian blur. For each pixel, we compute the absolute difference between the pixel in reference and the input image at the

same position with the following formula: $dif = |ref\_pixel - input\_pixel|$.

A Closing / Opening Kernel

The step here is composed of two sub-steps, one is the closing then the opening on the closing. Those are performed using two basic operations, erosion and dilation. The erosion consists of keeping the pixel with the highest value within the kernel. The dilation is the same operation but we are keeping the lowest value. In the case of the closing, we perform the dilation on the erosion and for the opening, we compute the erosion on the dilation.

Threshold

In addition to the closing/opening, we need to apply a threshold on the result. To do that we decided to use the Otsu method. This algorithm works by applying a histogram of the intensity values of the pixels in the image and then finding the threshold value that minimizes the variance between the two regions. This threshold value is then applied to the image to segment it into distinct regions. The Otsu method is one of the most popular and effective methods [1] for image processing.

Connected Component

For this part, we do a first pass to label every pixel with a 4-connectivity. When we encounter 2 neighbors labeled we choose the smallest and propagate this to the highest neighbor. After that, we retrieve all our different components and filter out the one with not enough pixels and the one with a peak pixel inferior to an arbitrary threshold.

4 Performance indicators used and why

For this project, we used time as the main performance indicator because the main purpose of using GPU programming is to accelerate computation time

[1] For reference you can read this article referencing to the methods of thresholding.

through the parallelization process. One of the over indicators is the number of iterations needed to compute the result. We choose those two because they help us to better illustrate the tradeoff between adding new iterations versus adding heavy operation.

5 Identification of performance bottlenecks

The main objective of this project is to iterate over our python, C++ CPU, and GPU implementations and to identify the bottlenecks of our algorithms. With some benchmarks, we could identify these bottlenecks of our different versions and try to improve over them.

We used Google Benchmarks [2] for the C++ CPU, C++ CPU openCV and C++ GPU implementations and the timeit builtin for the python implementation. The benchmark has been done on the different steps of the algorithm, which are the grayscale, the gaussian blur, the difference, the morphological closing and opening, the threshold, the connected component labeling and on the full pipeline for each version.

The specifications are the following :

- CPU : AMD® Ryzen™ 5-5600X @ 3.70GHz
- Run on (12 X 4641.35 MHz CPU s)
- CPU Caches:
  - L1 Data 32 KiB (x6)
  - L1 Instruction 32 KiB (x6)
  - L2 Unified 512 KiB (x6)
  - L3 Unified 32768 KiB (x1)
- RAM : 16 GO DDR4 3200 MT/s

[2] Google Benchmark is a library to benchmark code snippets, similar to unit tests.

Step	Python	C++ CPU	C++ CPU openCV
Grayscale	258571 ns	330781 ns	27312 ns
Gaussian Blur	408839 ns	19569900 ns	34410 ns
Difference	87941 ns	44839 ns	6242 ns
Closing/Opening	3120909 ns	196314319 ns	5166 ns
Threshold	44995 ns	59332 ns	4297 ns
Component	442273 ns	371191 ns	257539 ns
Full pipeline	8648994 ns	234961644 ns	591758 ns

Implementations Benchmark, results in nanoseconds

As we can see here, the naive C++ CPU version is the indisputable slowest version compared to the python openCV and C++ CPU openCV. A first analysis reveals that the application of the Gaussian Blur and the application of the morphological operation of closing and opening are the bottlenecks of this implementation.

A further analysis with the time percentage of the steps helped us to understand this hypothesis in a better view :

Step	Python	C++ CPU	C++ CPU openCV
Grayscale	6 %	0.1 %	8 %
Gaussian Blur	9 %	9 %	10 %
Difference	2 %	0.02 %	2 %
Closing / Opening	72 %	91 %	1.5 %
Threshold	1 %	0.02 %	1.2 %
Connected Component	10 %	0.1 %	77 %

Time percentage on Profiling : basic image (360 x 640)

This table is interesting because it shows some great insights and differences between the 4 versions. Most importantly it is useful to detect the bottlenecks in the steps.

Step	Python	C++ CPU	C++ CPU openCV
FPS	115	4	1700

Implementations Benchmark, FPS rate

6 Improvement over the GPU baseline

Over the creation of the GPU implementation, we discover some possible improvements along the process pipeline. From an algorithm point of view to increase the speed of our pipeline we can call every kernel inside a single function to avoid copying memory between host and device memory. In the same register, we could remove the save of each image after an operation. As we all know those IO operations are very costly like the copy from one to the other.

Another improvement could be the use of the GPU for the histogram computation to make it even faster. In this case, we should be careful as this will lead to atomic operations that lock the threads to avoid concurrency writing on the same address. This operations are known to be very costly.

From our benchmark, we see that one of the main depending process is the computation of the gaussian blur to speed up this part we could use a fast gaussian blur method such as the one described in the website[3]. The main idea between those methods is to apply 2D gaussian blur thanks to the kernel separable properties it expresses the 2D convolution as a combination of two 1D convolutions.

One of our other issues about the performance is with the morphological opening and closing to increase our speed we could use one of the implementations described in the paper[4]. The research paper talks about the performance of the Morard and Bartovsky algorithms in comparison of the classical CPU algorithms.

7 Summary table

[3] https://blog.ivank.net/fastest-gaussian-blur.html referencing the methods of fast blur

[4] https://hal.archives-ouvertes.fr/hal-00680904/document comparison between the methods of morphological opening/closing

8 Summary of the contributions of the members

Members	Python	C++ CPU	C++ GPU	Benchmarks	Report
Adrien	x	x	x	x	x
Alexandre	x	x	x	x	x
Sarah	x	x	x	x	x
Victor	x	x	x	x	x

Contributions summary