Optimization of the Sobel Operator with CUDA

Lingyi Song
Department of Automotive Engineering, Clemson University
International Center for Automotive Research (CU-ICAR)
2020 Summer Course Report: CPSC 6780 General Purpose Computation on GPU, Instructor: Prof. Shuangshuang Jin

Abstract

Edge detection is one of the most important paradigm of image processing. In Autonomous Driving and Advanced Driver-assistance Systems (ADAS), edge detection is very useful for detecting the lanes. Images contain millions of pixels and each pixel information is independent of its neighboring pixel. How to compute in a faster speed is an important problem. Hence this project puts to test the capability of Graphics Processing Unit (GPU) to compute in parallel for speeding up the Sobel filter for detecting edges. Performance and outputs of Sobel filter implemented in both Computing Processing Unit (CPU) and GPU has been compared, and an optimization comparison for GPU implementation is discussed.

1. Introduction

1.1 The motivation of this work

The Sobel operator is extensively used, mainly in edge detection algorithms for various applications. Also, it is a well-known first step in several computer vision and pattern recognition applications. The GPU has become one of the ideal solutions to accelerate data parallel computing. I choose to implement the Sobel detector application on parallel graphic processor for this project.

In fact, I used an implementation of the Sobel filter using CUDA. To be more specific, the implementation with CUDA is compared with the implementation with CPU in pixels, and with the OpenCV package. The optimization is described with different organization of dimensions of the threads and blocks for Sobel operator with CUDA.

The remainder of this report is organized as follows: first provide a brief review of the CUDA architecture and the Sobel filter detector. Then, it describes the programming model and how to optimize the speed. In the results, the execution time and final output are compared.

1.2 The challenges of this work

- Understanding and execution of the algorithm.
- Implementing the algorithm in different methods and platforms.
- Ensure that when implementing on GPU, the output image should not be worse than the output of CPU.
- Implementing the algorithm with CUDA for getting better speed while not losing the quality of the filter.

2. Methodology

2.1 Overview of CUDA and Sobel Operator

CUDA(Compute Unified Device Architecture) is a novel technology of general-purpose computing on the GPU. A certain processing flow of CUDA that need to be maintained as: Copy input image arrays from CPU memory to GPU; Load GPU program and execute, caching data on chip for performance; Copy result image arrays back from GPU memory to CPU memory to further manipulate or display the results.

The programmer determines the number of threads that best suits the given problem. The thread count along with the thread configurations are passed into the kernel. A block is an array of concurrent threads that execute the same thread program. The threading configuration is then passed to the kernel. Within the kernel, this information is stored as built-in variables. In short, the organizing of the dimensions of threads and blocks can dramatically influence the performance.

The Sobel operator is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is the corresponding gradient vector or the norm of this vector.

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the vertical and horizontal derivative approximations respectively, as (1) and (2), where * here denotes the 2-dimensional signal convolution operation.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \qquad (1)$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \qquad (2)$$

The x-coordinate is defined here as increasing in the "right"-direction, and the y-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using (3):

$$G = \sqrt{{G_x}^2 + {G_y}^2} \qquad (3)$$

Using this information, we can also calculate the gradient's direction:

$$\Theta = \arctan(\frac{G_y}{G_x}) \qquad (4)$$

In this case, for example, Θ is 0 for a vertical edge which is lighter on the right side.

2.2 Implementation Details

OpenCV package has been used for fundamental processing for images, such as read and write, transferring to grayscale. For all the methods, I first implement a 3x3 Gaussian Filter on the grayscale image of my input. This is for smoothing and reducing the image noises. The original image and the outputs of grayscale and filtered by Gaussian Filter are shown in Figure 3. The original image size is 599 x 393.

2.2.1 CPU Implementation

There are two implementations for CPU of Sobel operator. One is simply computing with CPU on pixels, and one is using the OpenCV function on image for computing. With the first one, I calculate the gradient in the two directions and then add their absolutes to get the integrated gradient, and then get the output. With the second one, I get the gradients with Sobel() function, and get the absolutes using convertScalAbs(), and integrate gradients with addWeighted().

2.2.2 GPU Implementation

Since the detection principle is through the convolution in two directions, we need to correctly index the position of each element centered at the target pixel, in the 3*3 filter, when implementing with CUDA. The image from CPU is a one-dimensional continuous memory, which increases the difficulty of indexing. Therefore, in the design of block and grid, I completely mapped the entire image to the grid. Each thread corresponds to a pixel.

However, in the first implementation of CUDA, I simply set both the thread number per block and block number per grid to 1. With this worst idea it can show the optimization potential for GPU implementation.

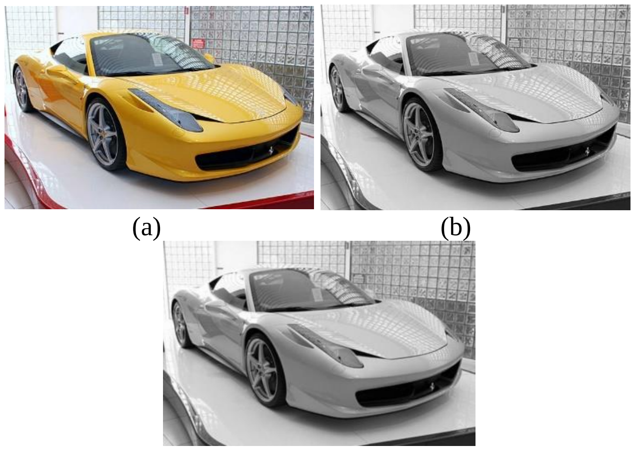

(a) (b) (c)

Figure 3: (a) Original; (b) Grayscale; (c) Gaussian Filter

2.2.3 GPU Implementation Optimization

As discussed in Section II, the dimensions of threads and blocks are very important for CUDA program. Contrast to the implementation in the last subsubsection, I reorganized the dimension with multiple threads per block and multiple blocks per grid. This is based on the constrain of the wrap number. They are set as follows:

blocks((int)((imgW+31)/32),(int)(g_imgHeight+31)/32);
threads(16, 16);

Also, the organization for threads index and computation details in kernel need to be modified correspondingly.

2.2.4 Execution Time

The execution time for CPU is provided by the clock() function in C, and cudaEvent() is used for recording the time for GPU execution.

3. Results

3.1 The Output Images

For image processing, it is very important to get the correct output. And in my project, the most important thing for the outputs is getting the same quality with different methods. The output images of CPU and GPU are shown in Figure 4.

We can see that the outputs from CPU and GPU are the same, no matter how the dimension of the CUDA kernel is organized. However, the outcome with OpenCV package is not totally the same with others. This is because of the predefined computing functions. In short, all the methods can produce satisfing output image, even the edges of the reflections on the car can be recognized easily. And the qulity of CPU and CUDA implementations is the same. We can conclud that it will not influence the output with CPU or GPU.

3.2 The Execution Time

The recoreded execution time of all the four programs is listed in Table I.

Table I: The execution time of different methods.

Implementation	Time (ms)
OpenCV package	1.700
CPU on pixels	12.545
GPU with single thread and block dimension	66.0277
GPU with organized thread and block dimension	0.5276

We can see that when computing with CPU, OpenCV functions can provide better speed. And when using single thread per block and single block per grid, the execution time is 60 times longer than using OpenCV package. But when organizing the dimension reasonably, the GPU implementation consumes 0.5276 ms for Sobel operator. Comparing with CPU, it speeds up the program by half the time; and it is more than 100 times faster than the bad organized dimension with GPU.

In conclusion, implementing Sobel operator with CUDA can accelerate the program by 3 times compared with CPU using OpenCV package, and more than 20 times compared with CPU operating on pixels. And the organization of the dimension of threads and blocks is very important for CUDA programs, better organization can result in 100 times difference in speed.

4. Future work

For future work, the implementation for other computer vision problems with GPU is also an attracting topic, such as Canny edge detection, and other filters. And there are just two different implementations for the CUDA program in this project, many modifications and researches could be conducted in the future, which would provide more interesting conclusions.

References

[1] A. Jain, A. Namdev and M. Chawla, "Parallel edge detection by SOBEL algorithm using CUDA C," 2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS), Bhopal, 2016, pp. 1-6, doi: 10.1109/SCEECS.2016.7509360.

[2] H. B. Fredj, M. Ltaif, A. Ammar and C. Souani, "Parallel implementation of Sobel filter using CUDA," 2017 International Conference on Control, Automation and Diagnosis (ICCAD), Hammamet, 2017, pp. 209-212, doi: 10.1109/CADIAG.2017.8075658.

[3] N. Kanopoulos, N. Vasanthavada and R. L. Baker, "Design of an image edge detection filter using the Sobel operator," in IEEE Journal of Solid-State Circuits, vol. 23, no. 2, pp. 358-367, April 1988, doi: 10.1109/4.996.

[4] DDGorade/Sobel-Edge-Detection-Using-CUDA, GitHub, a ccessed 1 July 2020,<https://github.com/DDGorade/Sobel-Edge-Detection-Using-CUDA/blob/master/Sobel_Edge_8Bit.cu >

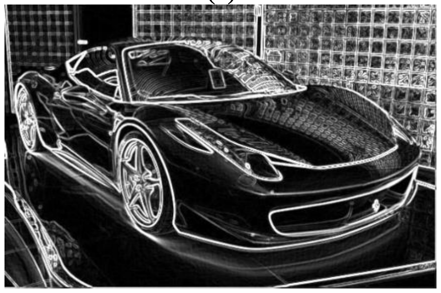
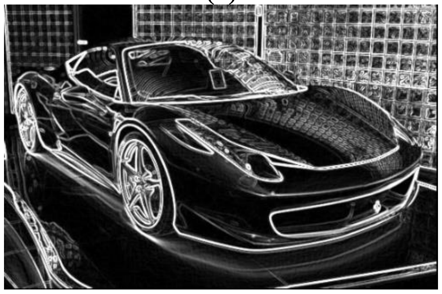
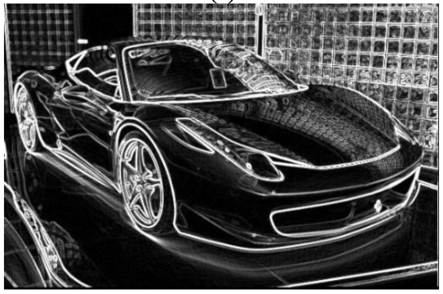
(a)

(b)

(c)

(d)

Figure 4: Output images. (a) is the output of OpenCV functions. (b) is the output of CPU on pixels. (c) is the output of GPU with single thread/block and single block/grid. (d) is the output of GPU with organized number of threads and blocks.