Edge detection filtering using CUDA

Student:
Eng. Cătălin-Aurelian Ciocîrlan

Project Advisor:
SrLec. PhD. Eng. George Valentin Stoica

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

May 2023

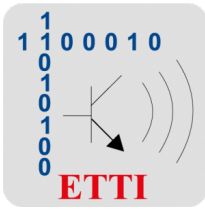

Table of contents

Theoretical overview

Problem formulation

Definition

- Edge detection is a technique of image processing used to identify points in a digital image with discontinuities. These points where the image brightness varies sharply are called the edges (or boundaries) of the image.
- An edge can be detected by analyzing the first derivative of the intensity profile, taken perpendicular to the edge.

Applications

- Identifying objects
- Measuring objects
- Segmenting images

Problem formulation

Figure 1: Original image

Figure 2: Sobel edges

Figure 3: Canny edges

Sobel filter

Description

- For this project I have used two algorithms, the first being a Sobel filter, which is a linear discrete differentiation operator that computes the local gradient at any point in the image.
- The operator uses two convolution kernels, one for each main direction, that can be decomposed as the products of an averaging and a differentiation kernel, in order to compute the gradient with smoothing, as to reduce noise amplification.

Sobel filter

Convolution kernels

- Horizontal gradient kernel:

$$K_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{1}$$

- Vertical gradient kernel:

$$K_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

Sobel filter

If we define I as the source image, and G_x and G_y are two images which at each point contain the horizontal and vertical derivative approximations respectively, the algorithms proceeds as follows:

1. Compute horizontal gradient:
$$G_x = I * K_x \tag{3}$$
2. Compute vertical gradient:
$$G_y = I * K_y \tag{4}$$
3. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:
$$G = \sqrt{G_x^2 + G_y^2} \tag{5}$$

Canny filter

Description

- The second algorithm used is a Canny filter, which uses a Sobel filter in order to extract the gradient, as well as the orientation of the respective derivative.
- The filter then applies non-maximum suppression to the gradient image to thin out the edges and remove any weak or spurious edges that may have been detected.
- Finally, the filter applies hysteresis thresholding to identify the strongest edges in the image.
- The output of the Canny filter is a binary image, where the edges of objects in the original image are represented as white pixels, and the background is represented as black pixels.
- Its greatest advantage are the thin edges, which are one pixel wide, even though it requires a greater computational power in order to achieve this result.

Canny filter

If we define I as the source image, G_x and G_y are two images which at each point contain the horizontal and vertical derivative approximations respectively and θ represents the gradient orientation of the pixel, the algorithms proceeds as follows:

1. Compute Sobel filter output:
$$G = Sobel(I) \tag{6}$$
2. Calculate gradient orientation based on G_x and G_y. The direction is rounded to one of four possible angles (namely 0, 45, 90 or 135):
$$\theta = arctan\left(\frac{G_y}{G_x}\right) \tag{7}$$
3. Normalize gradient so that it has integer values between 0 and 255:
$$G_{norm} = \left\lfloor \frac{G_{i,j} - \min\limits_{i,j} G}{\max\limits_{i,j} G - \min\limits_{i,j} G} \cdot 255 \right\rfloor \tag{8}$$

where $G_{i,j}$ is the gradient in any point of the image;

Canny filter

4. Non-maximum suppression: the algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions, in order to thin the edges;
5. Double thresholding: the double threshold step aims at identifying 3 kinds of pixels: strong, weak, and non-relevant. Strong pixels are pixels that have an intensity so high that we are sure they contribute to the final edge. Weak pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection. Other pixels are considered as non-relevant for the edge;
6. Edge tracking by hysteresis: based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one. The output of this step represents the final output of the filter.

Implementation

Implementation details

Versions

- Four different versions of both algorithms were realised: OpenCV, OpenCV CUDA, native C++ and CUDA.
- Both OpenCV implementations use the OpenCV API in order to compute the edges, while the other two are implemented from scratch.
- The code is available on my personal Github repository.

Implementation details

CUDA implementation details

- For the CUDA implementation of the Sobel filter, four different variations where realised:
  1. V1 – Pixel level thread 1: a thread computes the gradient for a single pixel, the grid is divided in *image height* number of blocks, of *image width* number of threads. Used for images with a width of maximum 1024 pixels.
  2. V2 – Pixel level thread 2: a thread computes the gradient for a single pixel, the grid is divided into blocks of 8 x 8 threads, the number of blocks covering the whole image. Used for images with a width larger than 1024 pixels. The number of threads per block was chosen empirically so that it gave the best performance.
  3. V3 – Region level thread 1: a thread computes the gradient for an entire region of the image, the grid consists of 1024 blocks with a single thread.
  4. V4 – Region level thread 2: the same as the previous one, but the respective region that a thread computes on is transferred to the shared memory so that the data is closer to the CUDA core.
- For the CUDA implementation of the Canny filter, only the first two variants were realised.
- The convolution kernels K_x and K_y were stored in the constant memory, given the fact that these variables do not change and they are used by every thread. Every other array is stored in global memory, with the exception of the image regions of Sobel V3.

Results and comparisons

Specifications and details

System specifications

- CPU: Intel i7-7700HQ, 2.80 GHz
- RAM: 24 GB, 2.4 GHz
- GPU: NVIDIA GeForce GTX 1050 Ti, 4 GB, 768 CUDA cores, 1.62 GHz core speed, 7 GHz memory speed

Measurement details

- For the Sobel Filter, the computations were repeated 100 times, while for the Canny filter 50 times, to ensure a correct time measurement.
- For every version, the time measurement takes into consideration all computations and internal allocations (only those needed in the functions, not the allocation of the input and output variables, which is done outside of the computation loop).
- All time is measured in milliseconds.
- Speed-up (S) is first computed between the OpenCV implementations, then between the C++ version and the respective CUDA counterpart.
- CUDA 11.6 was used.

Sobel measurements

Image size	OpenCV [ms]	OpenCV CUDA [ms]	S
512 x 512	522	478	**1.092**
512 x 1024	845	618	**1.367**
1024 x 1024	1465	748	**1.958**
1024 x 2048	2309	923	**2.501**
2048 x 2048	3932	1279	**3.074**

Table 1: Sobel OpenCV time measurements and speed-up

Image size	C++ [ms]	CUDA V1/V2 [ms]	$S_{V1/V2}$	CUDA V3 [ms]	S_{V3}	CUDA V4 [ms]	S_{V4}
512 x 512	555	7.401	**74.990**	27.099	**20.480**	45.711	**12.141**
512 x 1024	1186	9.959	**119.088**	84.018	**14.116**	156.628	**7.572**
1024 x 1024	2636	18.445	**142.911**	300.179	**8.781**	516.466	**5.104**
1024 x 2048	5477	83.736	**65.408**	597.899	**9.160**	1024.18	**5.348**
2048 x 2048	9154	160.536	**57.021**	1333.62	**6.864**	2090.73	**4.378**

Table 2: Sobel C++ and CUDA time measurements and speed-up

Sobel Measurements

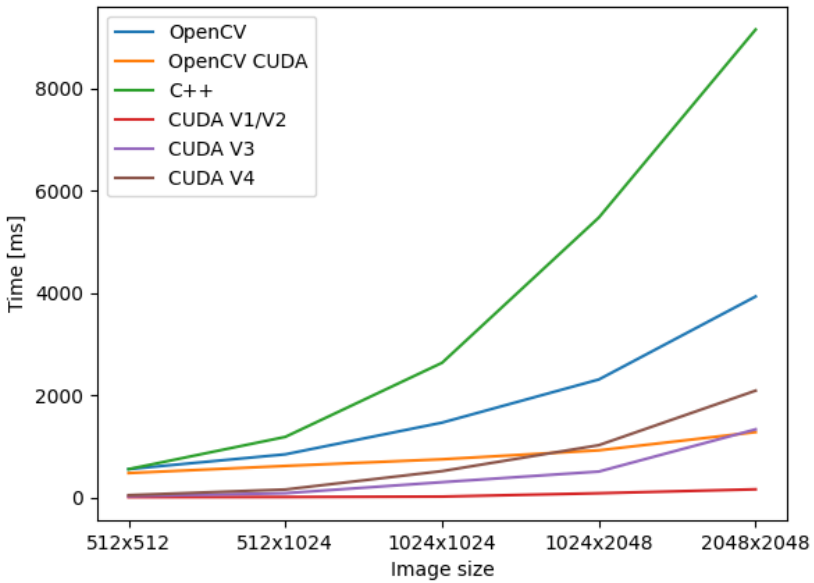

Figure 4: Sobel time measurements

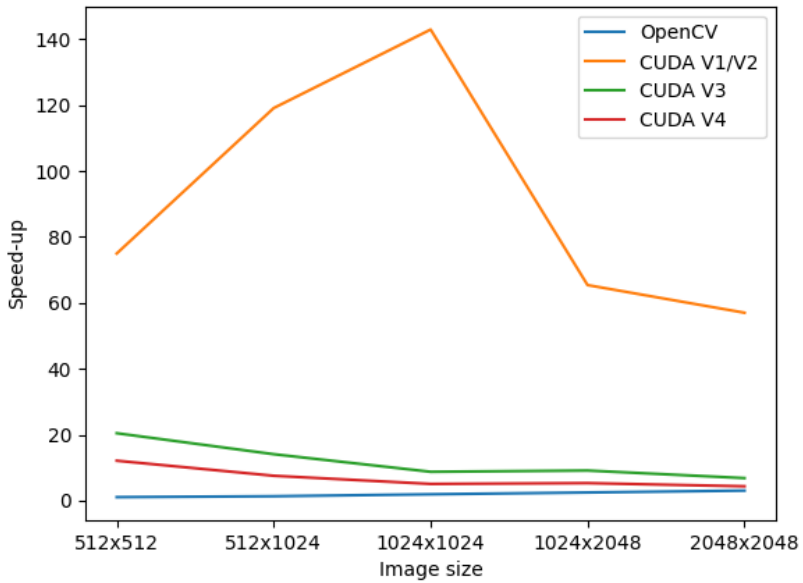

Figure 5: Sobel speed-up measurements

Sobel Measurements

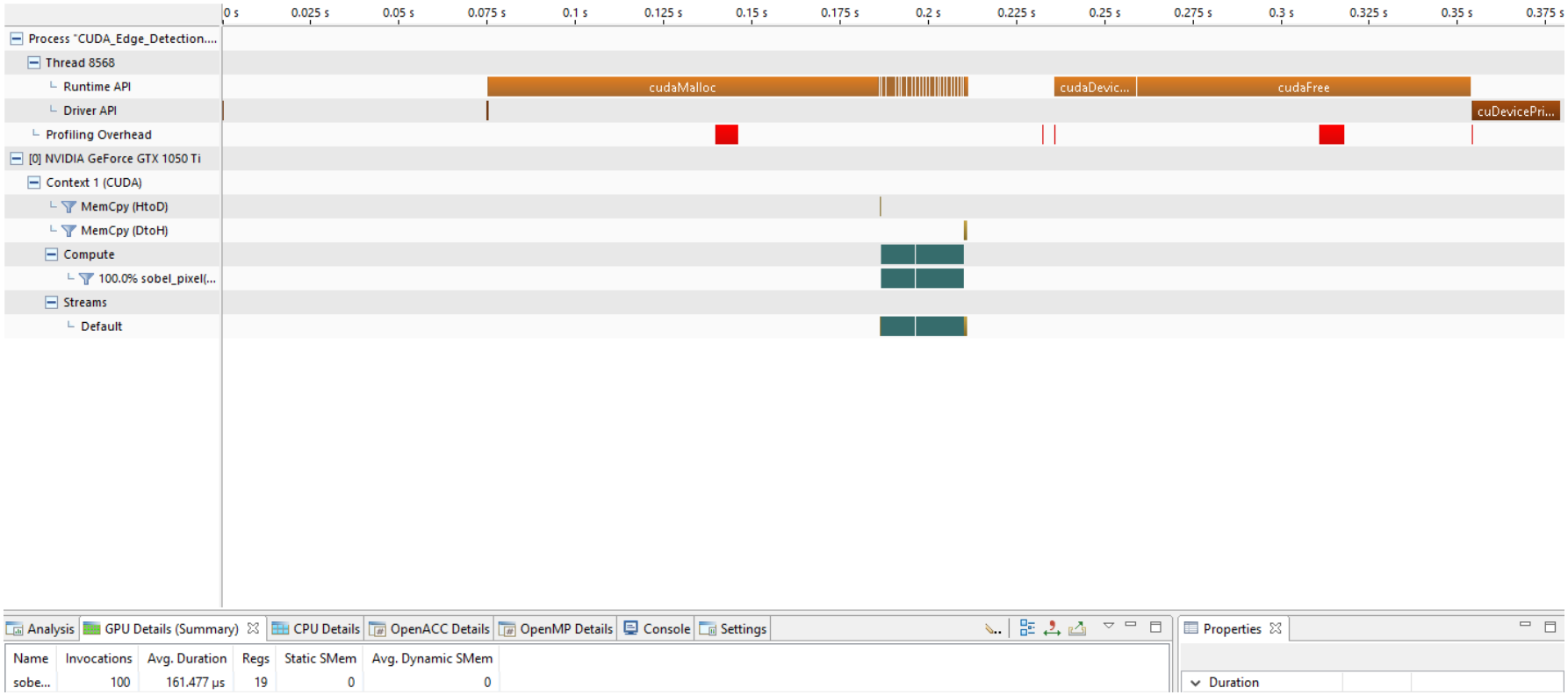

Figure 6: Nvidia Profiler output for the Sobel filter, using an image of size 1024 x 1024. As it can be seen, most of the duration of the program is spend on allocating data and moving it from host to device and back, while an average kernel launch take only around 160 μs s. Each core uses 19 registers to store local data.

Canny measurements

Image size	OpenCV [ms]	OpenCV CUDA [ms]	S
512 x 512	223	100	**2.230**
512 x 1024	585	257	**2.276**
1024 x 1024	598	221	**2.706**
1024 x 2048	747	397	**1.882**
2048 x 2048	1419	598	**2.373**

Table 3: Canny OpenCV time measurements and speed-up

Image size	C++ [ms]	CUDA V1/V2 [ms]	$S_{V1/V2}$
512 x 512	1184	129.789	**9.122**
512 x 1024	2278	183.156	**12.437**
1024 x 1024	4530	287.114	**15.778**
1024 x 2048	7838	688.238	**11.389**
2048 x 2048	15585	1186.35	**13.137**

Table 4: Canny C++ and CUDA time measurements and speed-up

Canny Measurements

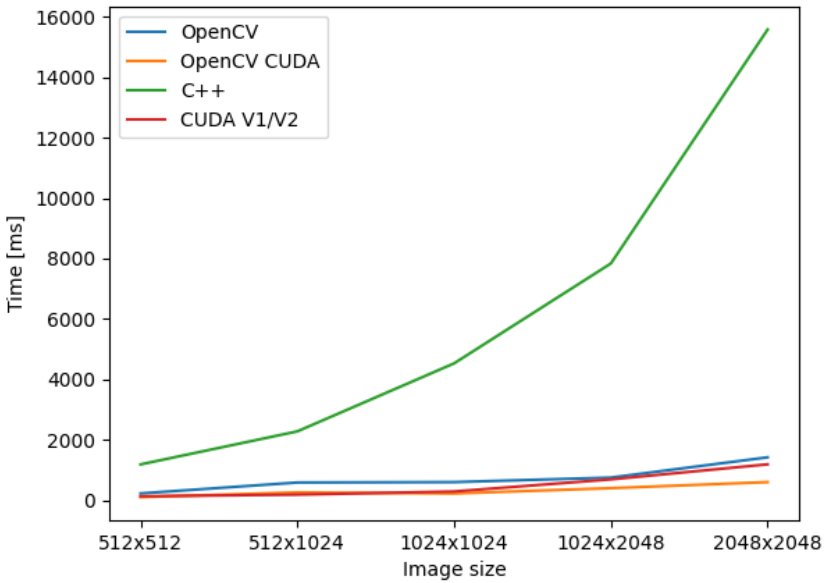

Figure 7: Canny time measurements

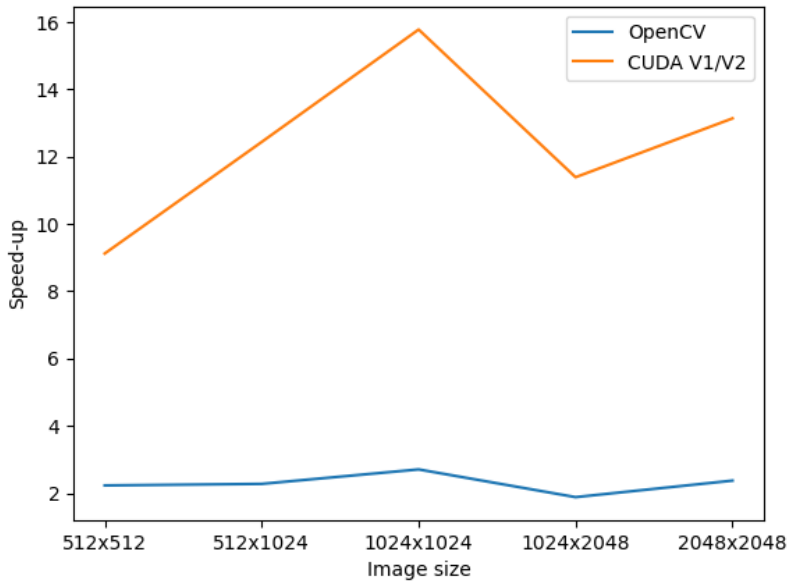

Figure 8: Canny speed-up measurements

Canny Measurements

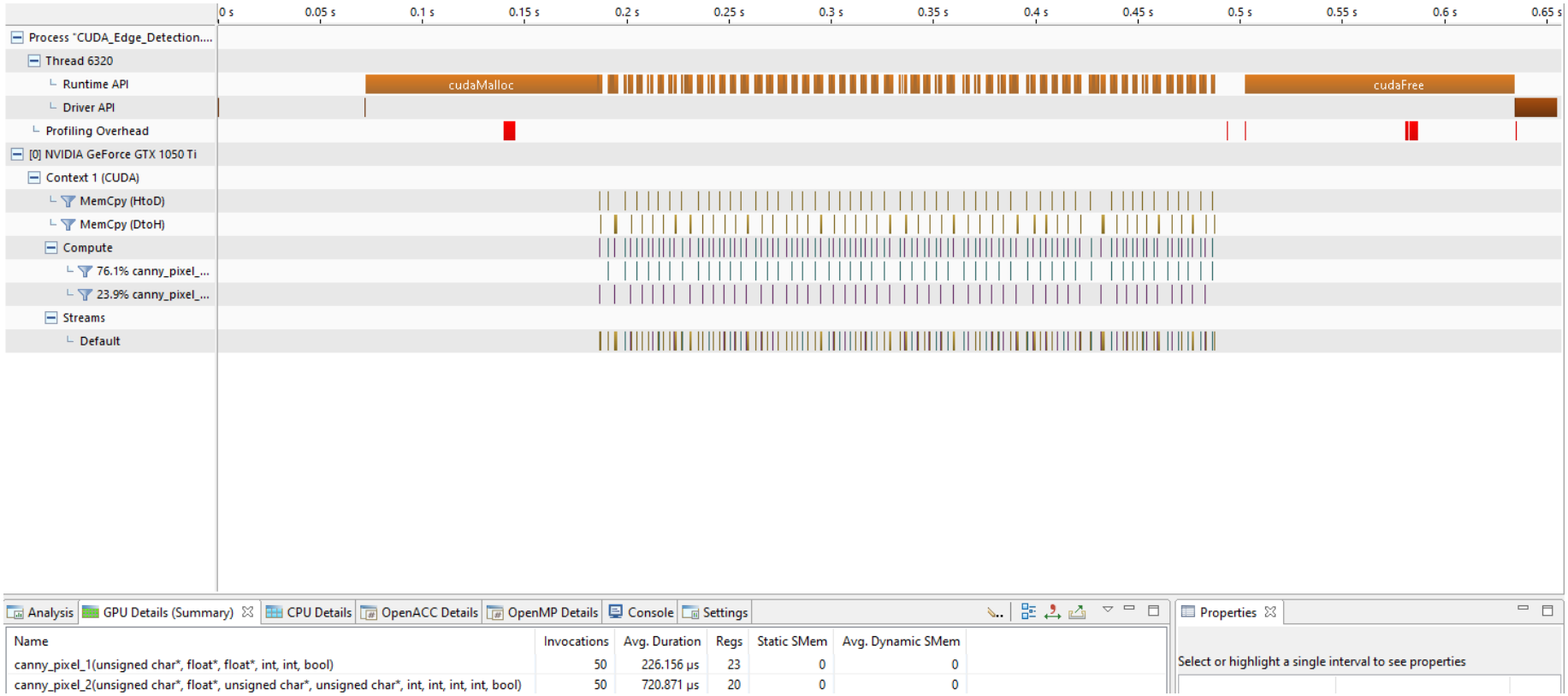

Figure 9: Nvidia Profiler output for the Canny filter, using an image of size 1024 x 1024. As it can be seen, most of the duration of the program is spend on allocating and moving the gradient matrix in order to be normalized, which represents the bottleneck of my algorithm. Each core uses 20 registers to store local data.

Block size comparison

Threads per block	Sobel CUDA V2 [ms]	Canny CUDA V2 [ms]
1 x 1	2112.130	3720.100
2 x 2	572.060	1760.190
4 x 4	164.293	1296.220
8 x 8	154.792	1199.530
16 x 16	309.336	1377.932
32 x 32	493.766	1470.85

Table 5: Comparison of the effects of choosing a different block size (and by this, as described earlier, also a different grid size) to launch a kernel. Measurements realised on a 2048 x 2048 image. With the color red is highlighted the size with the best performance found.

Conclusions

Conclusions

Conclusions

- Pure CUDA implementations for the Sobel filter are much faster than the OpenCV CUDA one, as boilerplate code of the API is avoided.
- Pure CUDA implementations for the Canny filter are performing worse than the OpenCV CUDA one, even though the acceleration is greater. This is because the bottleneck of my algorithm normalizes the gradient on the CPU, instead of the GPU, and the transfer of data between the device and the host has a significant effect. An ideal implementation should include a reduction algorithm performed on the device in order to find the minimum and maximum values of the matrix, which are used for the normalization.
- Pixel level implementations are faster than region level ones.
- Maximum acceleration is achieved for images with a large height and a width of 1024.
- Best block size for V2 variants of the algorithms are 8 x 8 threads, with a grid size calculated according to the input image dimensions.

Thank you for your attention!