Canny edge detection with CUDA

Lecture

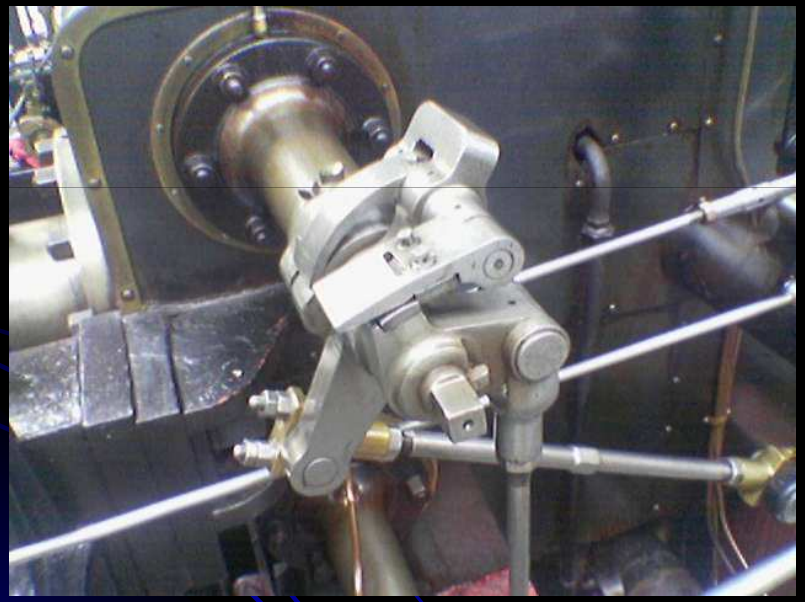
Alexey Abramov
abramov _at_ physik3.gwdg.de

Georg-August University, Bernstein Center for Computational Neuroscience,
III Physikalisches Institut, Göttingen, Germany

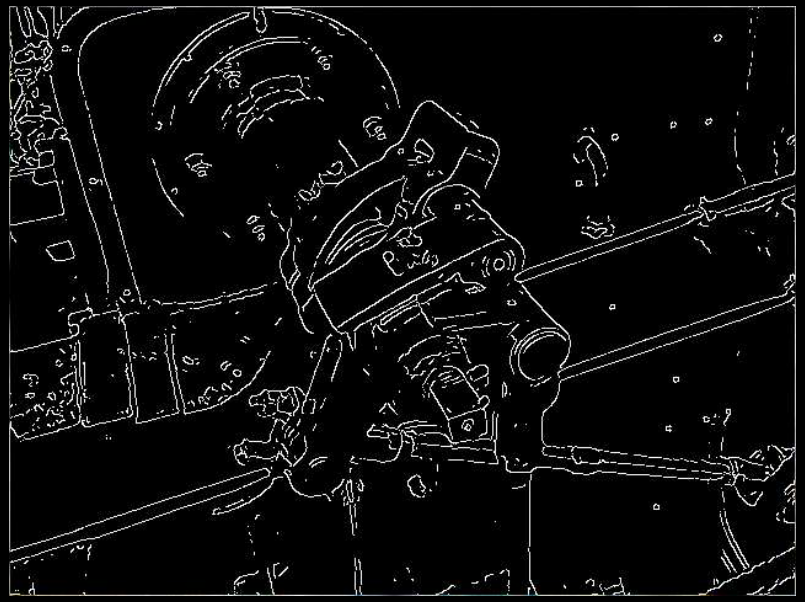
Canny edge detector

The **Canny edge detection** operator was developed by John F. Canny in 1986 and uses a multi-stage algorithm to detect a wide range of edges in images.

Original image



Output of the Canny edge detector



Canny edge detector

Canny defined optimal edge finding as a set of criteria that maximize the probability of criteria that maximize the probability of detecting true edges while minimizing the probability of false edges.

The main stages of the Canny algorithm:

- Convert to grayscale
- Noise reduction by filtering with a Gaussian blurring filter
- Compute gradient magnitude and angle
- Relate the edge gradients to directions that can be traced
- Tracing of valid edges
- Hysteresis thresholding to eliminate breaking up of edge contours

Conversion and noise reduction

It is inevitable that all images taken from a camera will contain some amount of noise. To prevent that noise is mistaken for edges, noise must be reduced. Therefore the image is first smoothed by applying a Gaussian filter.

$$B = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

σ = 1.4

Finding gradients

An edge in an image can point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. Gradients at each pixel in the smoothed image are determined by applying what is known as the **Sobel-operator**.

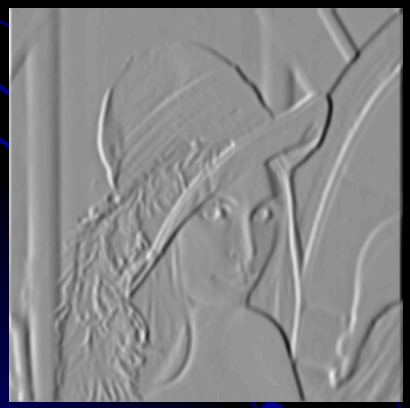
$$K_{GX} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_{GY} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

gradient magnitude
and
gradient's direction

$$|G| = \sqrt{G_x^2 + G_y^2}$$

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

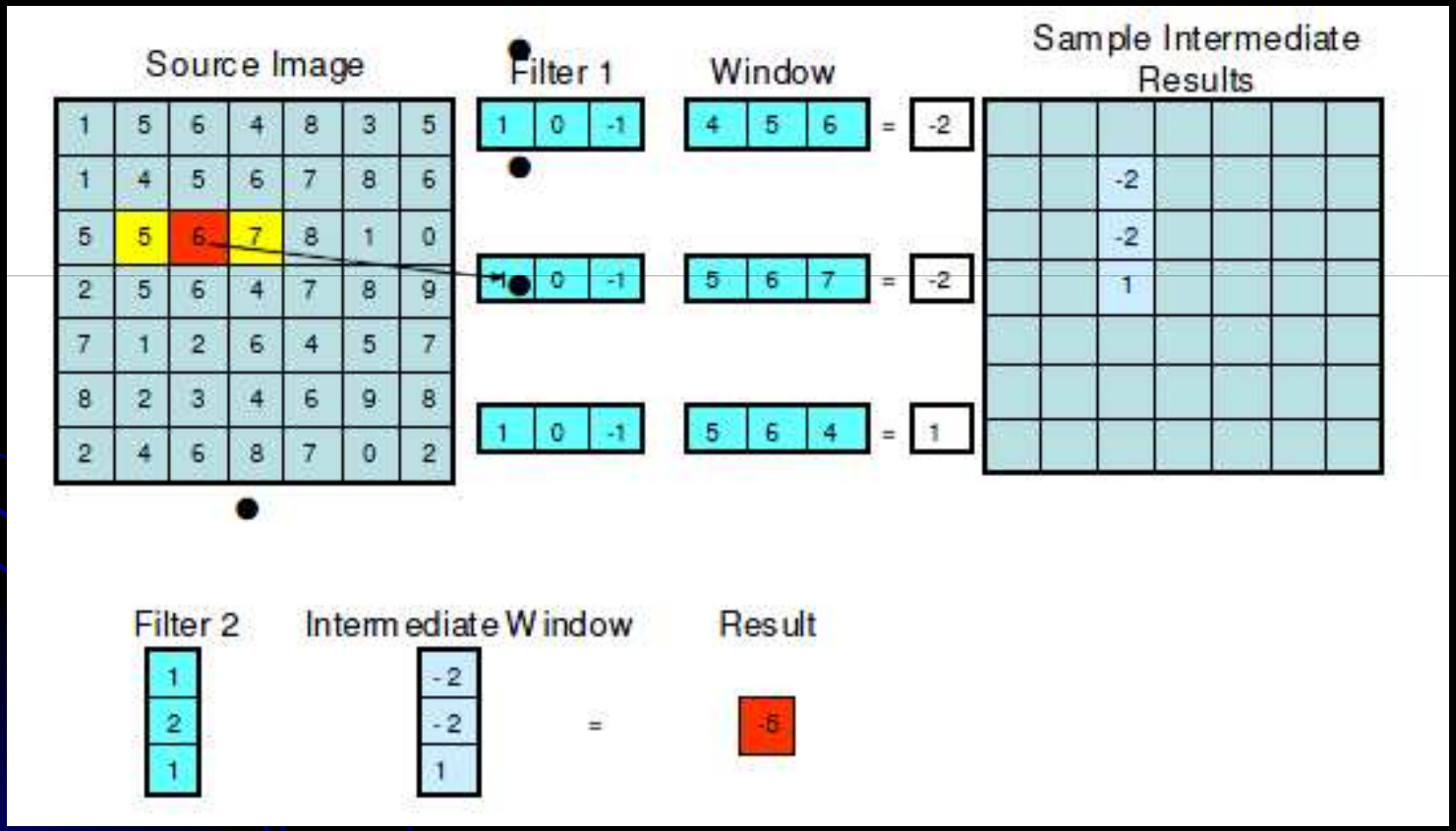
Vertical edges



Horizontal edges

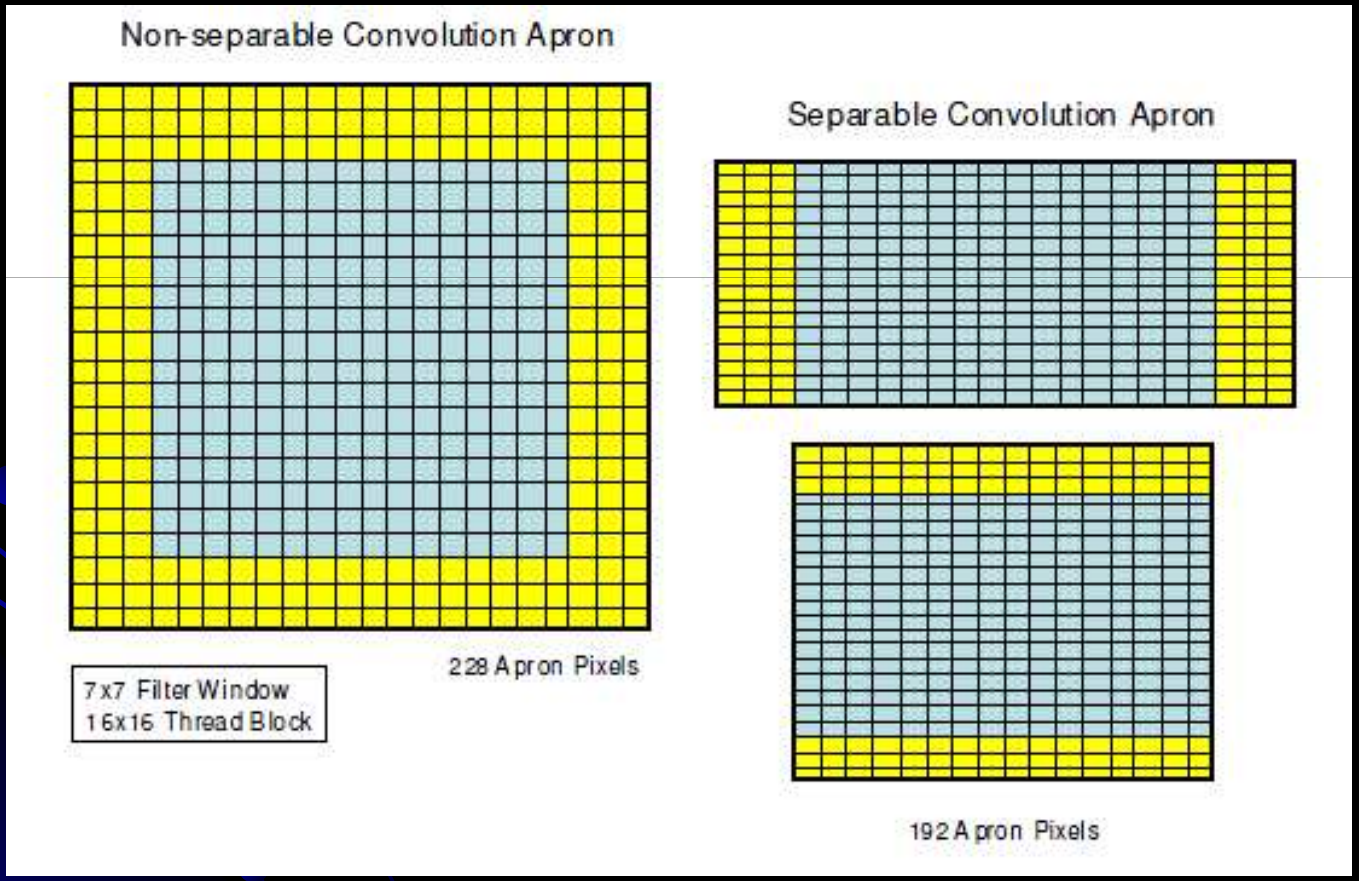
Gaussian and Sobel filtering on a GPU

Sobel and Gaussian filters are **separable functions**. Generally, a non-separable filter of window size M x M computes M^2 operations per pixel, for separable filters the cost would be reduced to computing M + M = 2*M operations.



Gaussian and Sobel filtering on a GPU

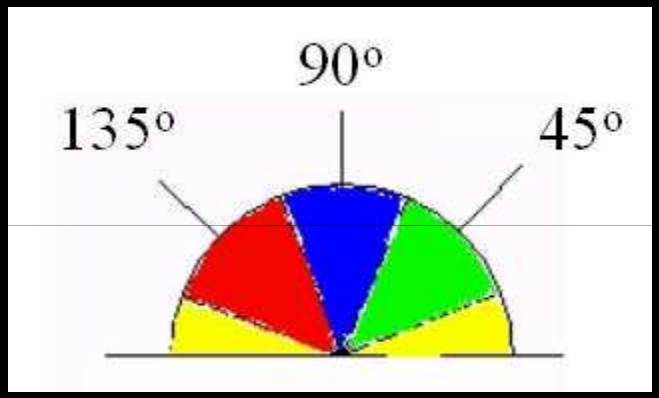
Apron pixels are necessary because convolutions near the edge of a thread block will have to access pixels normally loaded by adjacent thread blocks. Since shared memory is local to individual thread blocks, each block loads its own set of apron pixels before processing.



Edge direction angle

Once the edge direction is known, the next step is to relate the edge direction to a direction that can be traced in an image. The edge direction is rounded to one of four angles representing vertical, horizontal and the two diagonals (0, 45, 90 and 135 degrees).

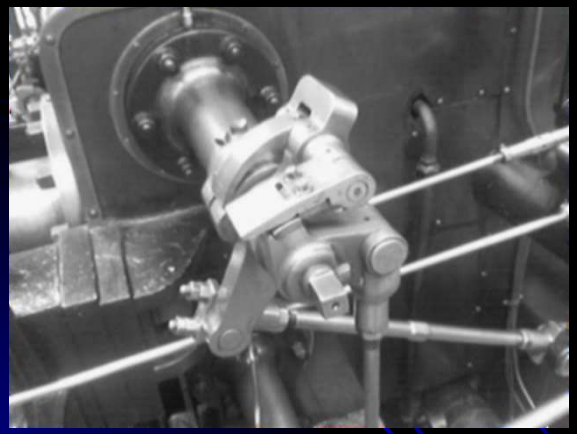
x	x	x	x	x
x	x	x	x	x
x	x	a	x	x
x	x	x	x	x
x	x	x	x	x



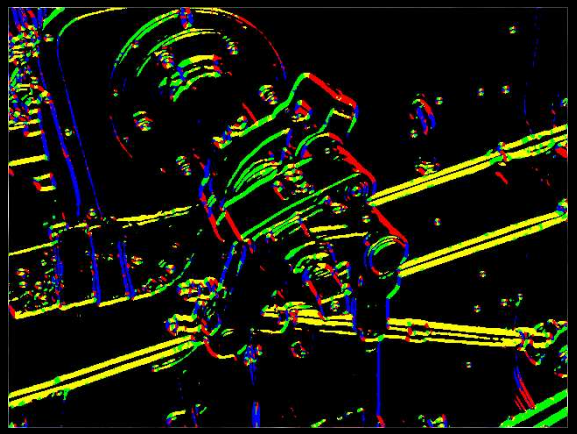
So now the edge orientation has to be resolved into one of these four directions depending on which direction it is closest to.

Non-maximum suppression

After the edge directions are known, non-maximum suppression now has to be applied. It is used to trace along the edge in the edge direction and suppress any pixel value (sets it equal to zero) that is not considered to be an edge. This will give a thin line in the output image keeping only those pixels on an edge with the highest gradient magnitude.



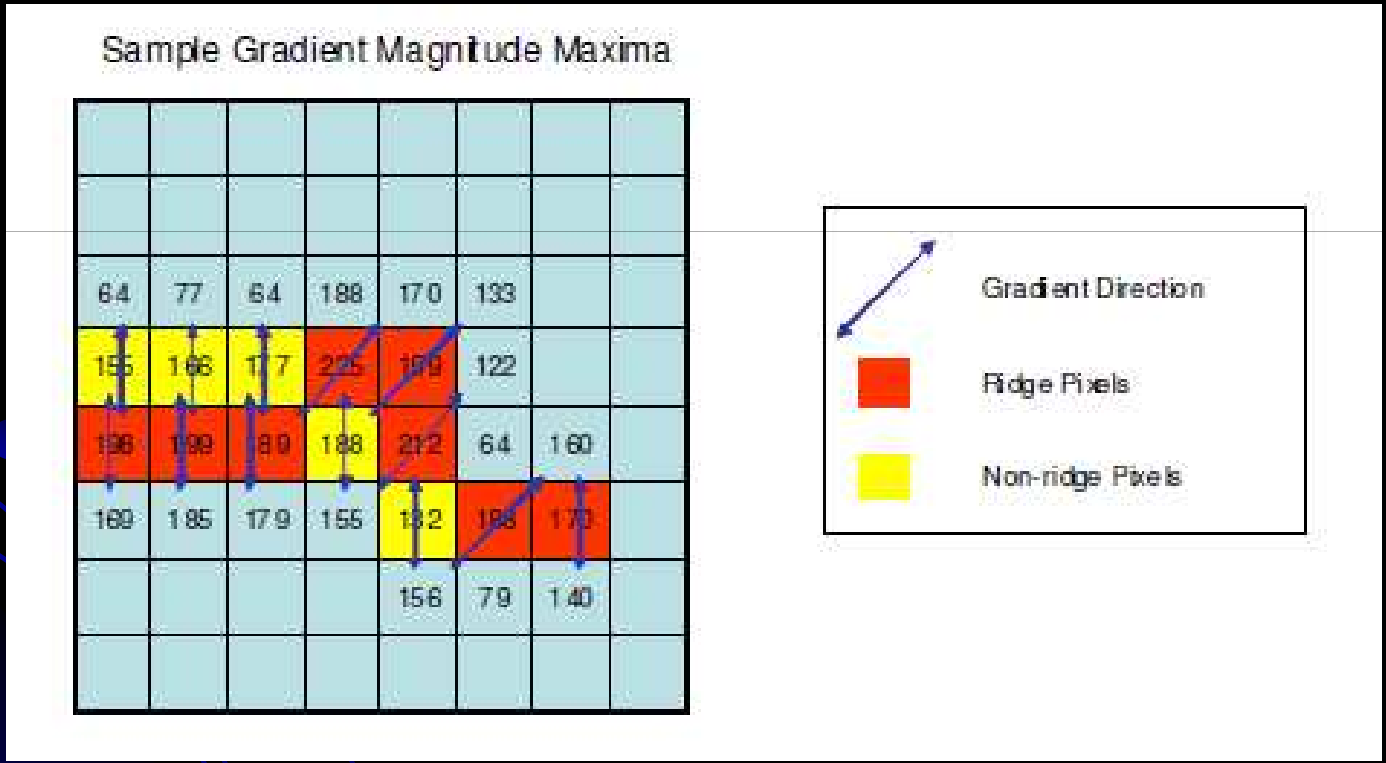
A binary edge map derived from the Sobel operator



Non-maximum suppression

Non-maximum suppression

Ridge pixels are defined as the pixels with gradient magnitudes greater than both of its adjacent pixels in the gradient direction. The function also requires a 1 pixel wide apron around each thread block since pixels along the perimeter have directional configurations extending outside of the normal pixel group range of the thread block.



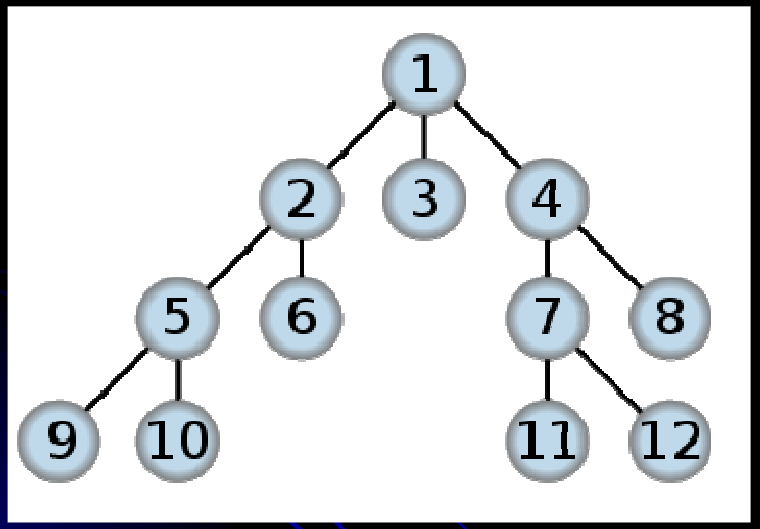
Hysteresis thresholding

Finally, hysteresis is used as a means of eliminating streaking. Streaking is the breaking up of an edge contour caused by the operator output fluctuating above and below the threshold. Instead of choosing a single threshold, two thresholds t_{high} and t_{low} are used. For a pixel **(x,y)** :

- if $G_{xy} < t_{low}$ – discard the edge (write out black)
- if $G_{xy} > t_{high}$ – keep the edge (write out black)
- if $t_{low} < G_{xy} < t_{high}$ and any of its neighbors in a 3 x 3 region around it have gradient magnitudes greater than t_{high} – keep the edge (write out white)
- if none of the current pixel's neighbors have high gradient magnitudes but at least one falls between t_{high} and t_{low}, search the 5 x 5 region to see if any of these pixels have a magnitude greater than t_{high}. If so, keep the edge (write out white)
- Else, discard the edge (write out black)

Breadth-first search (BFS)

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

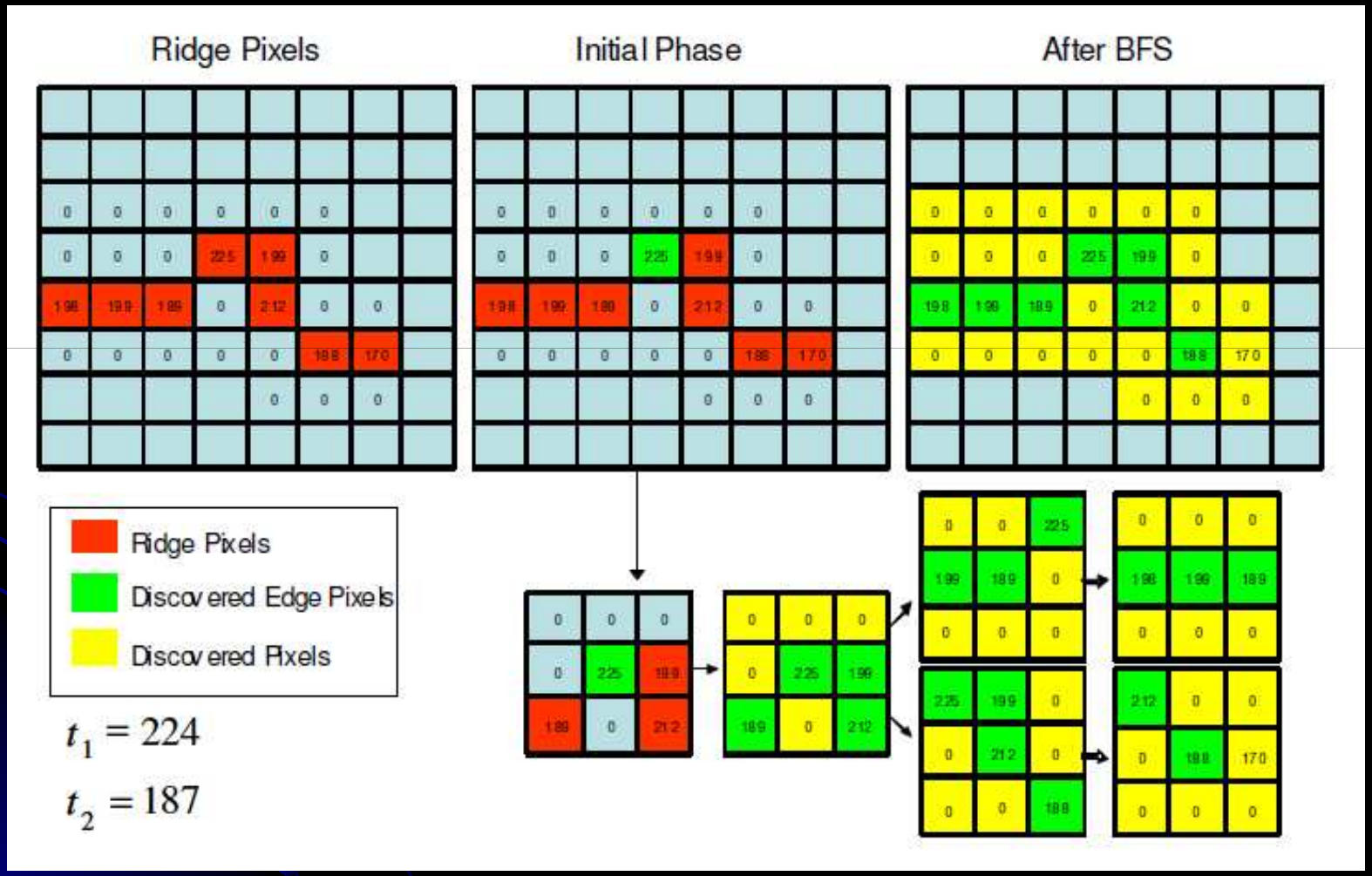


Pseudocode:

```
1 procedure BFS(Graph,source):
2     create a queue Q
3     enqueue source onto Q
4     mark source
5     while Q is not empty:
6         dequeue an item from Q into v
7         for each edge e incident on v in Graph:
 8            let w be the other end of e
9             if w is not marked:
10                mark w
11                enqueue w onto Q
```

Hysteresis thresholding

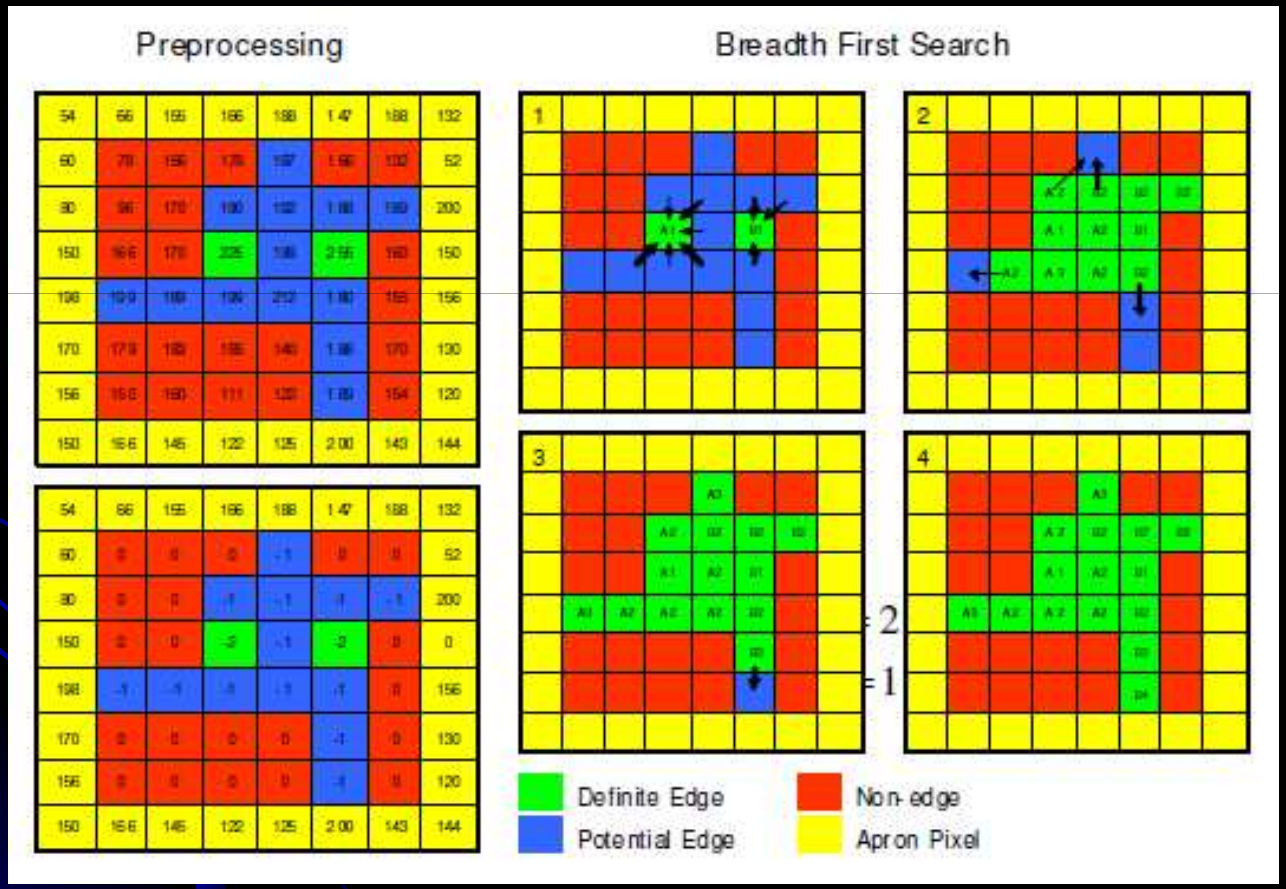
Pixels over the t_1 threshold are added to the queue. BFS iterates through undiscovered adjacent pixels and adds to queue if pixel value is over t_2.



CUDA hysteresis and BFS

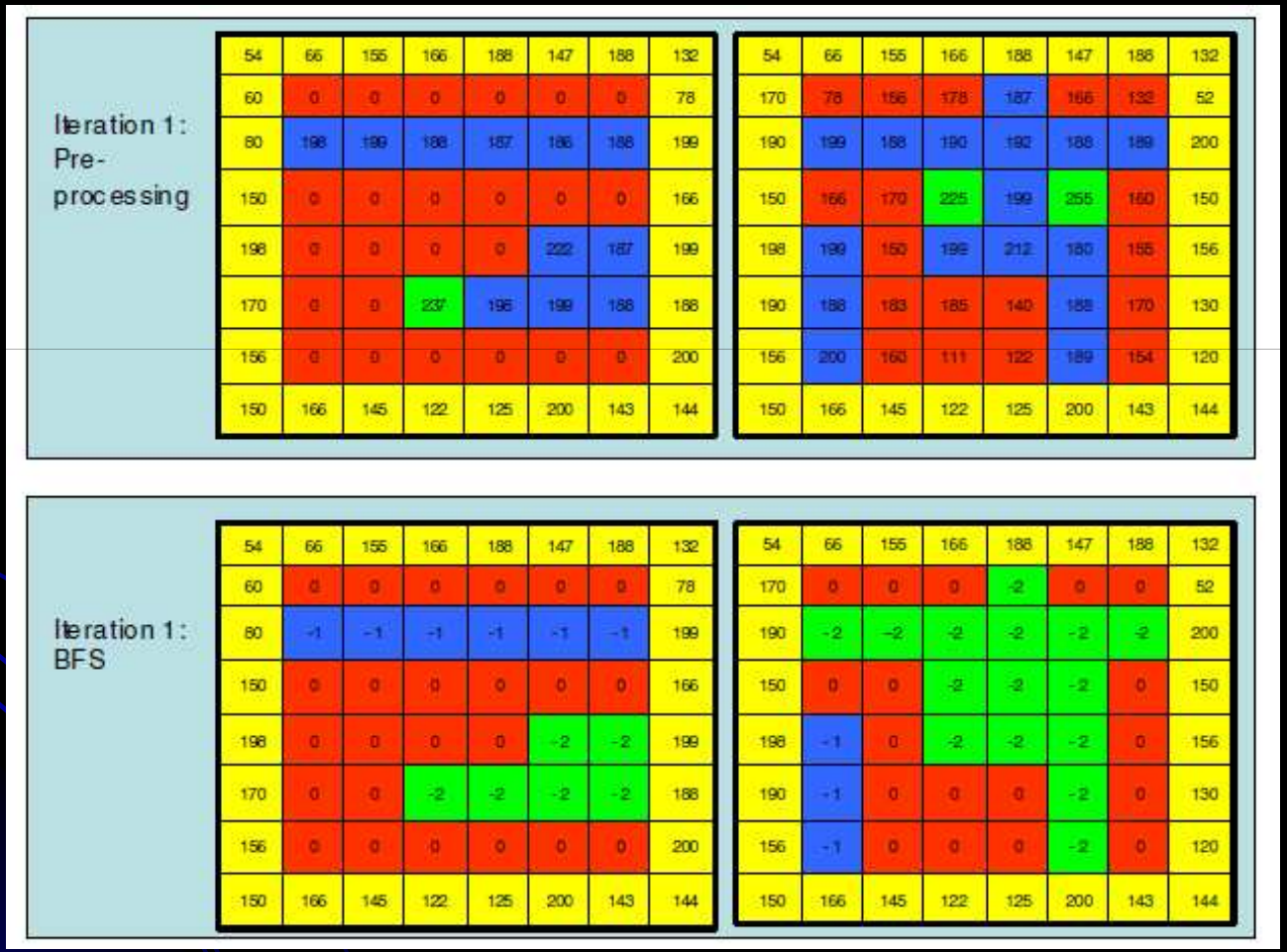
Each thread performs a BFS on a group of pixels. Pixel visited/processed states are tracked in shared memory and accessible to all threads in the thread block.

-2 definite border **-1** potential border **0** non-edge

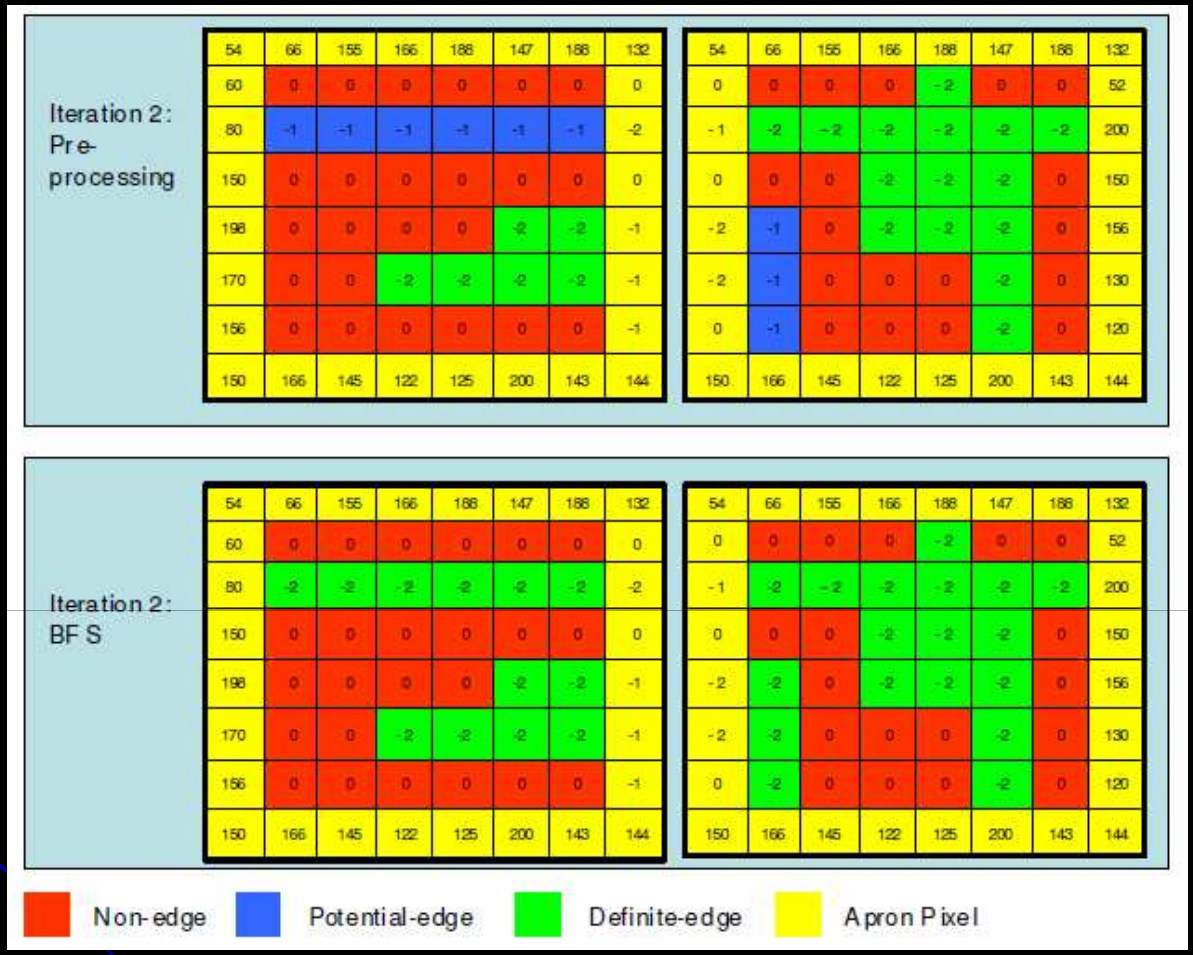


CUDA hysteresis and BFS

Multi-pass hysteresis and connected components of two adjacent thread blocks. After the first iteration, two pixel chains remain disconnected because two thread blocks performed processing. After the second iteration, the chains connect because of updated apron pixels.



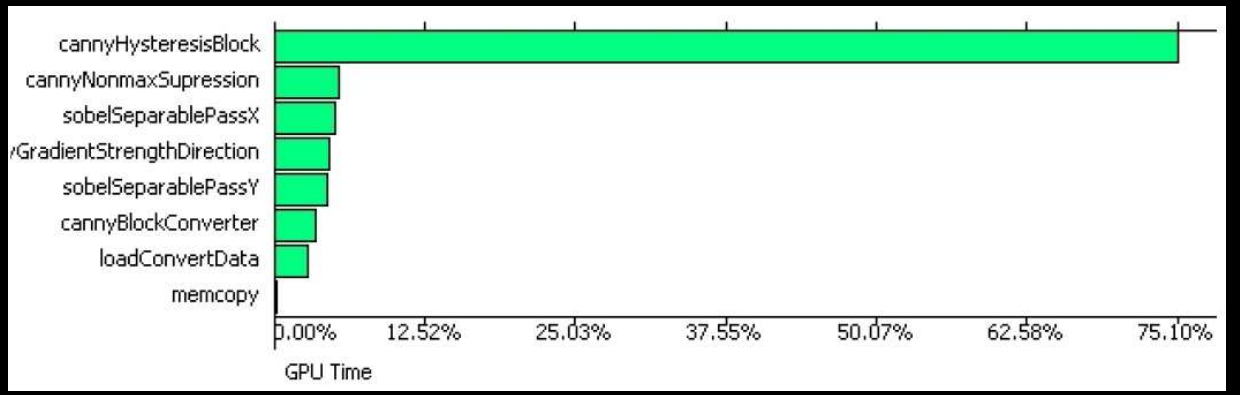
CUDA hysteresis and BFS



The main issue of hysteresis and connected components in CUDA is the necessity for inter-thread block communication. Although threads within a thread block can be synchronized, threads in different blocks are not.

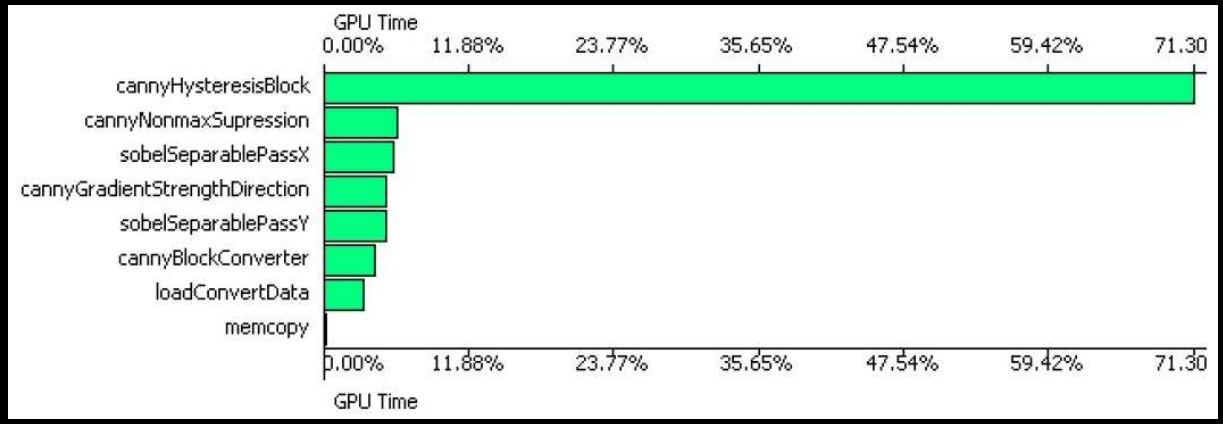
Results

Image Size	CUDA (ms)	OpenCV	Speedup
256× 256	1.82	3.06	1.681
512× 512	3.40	8.28	2.435294
1024× 1024	10.92	28.92	2.648352
2048× 2048	31.46	105.55	3.353465
3936× 3936	96.62	374.36	3.874560



Results

Image Size	CUDA (ms)	OpenCV	Speedup
256x240	1.63	3.92	2.405
512x496	3.60	12.25	3.403
1024x992	12.93	42.70	3.302
2048x1968	31.46	142.70	2.839
4096x3936	153.74	454.60	2.957



Bibliography

- **J.F. Canny**, “A computational approach to edge detection”
- **Y. Roodt**, “Image processing on the GPU: Implementing the Canny edge detection algorithm”
- **Y. Luo**, “Canny Edge Detection on NVIDIA CUDA”

Thank you for your attention !

QUESTIONS ?

Göttingen, 11.03.2011