Edge Detection - Canny Filter Implementation

Jonathan Lam

May 10, 2021

1 Abstract

Edge detection is a common and important task in image processing. One of the standard edge detection algorithms is the Canny edge detector, a six-stage algorithm. We implement the the filter in CUDA C++ based on Luo and Duraiswami [1] and a baseline CPU version. This paper describes the theory behind the Canny filter, a naive approach covering all six stages, and several optimizations based on Luo and Duraiswami's work, along with relevant code samples. We evaluate the performance of the different implementations compared to one another on several test images.

2 Introduction

According to the MATLAB documentation[1], "Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness. Edge detection is used for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision. Common edge detection algorithms include Sobel, Canny, Prewitt, Roberts, and fuzzy logic methods."

We had originally decided to implement the Sobel filter, which comprises of two (a horizontal and vertical) 3×3 differential filters. However, this proved to be too simple for the purposes of this project, so we decided to attempt an implementation of the Canny algorithm, which includes the Sobel filter as the second step.

The Canny filter was developed by John Canny in 1986 [2]. It comprises six steps:

1. turn the image into a single channel (grayscale)
2. applying a smoothing (denoising) filter
3. finding the intensity gradients and directions
4. non-maximum edge suppression (i.e., edge thinning)

[1]`https://www.mathworks.com/discovery/edge-detection.html`

5. double-thresholding the remaining edges
6. edge tracking via hysteresis

The following sections will review the intuition and general method for each step. A visual summary of the steps can be found in Figure 1; this is our reproduction of the example provided on the Wikipedia page for the Canny filter[2].

2.1 Grayscale

Edge detection depends only on the brightness gradients of the image, and not on the color. If we begin with a color image, we must begin by converting it to some grayscale variant. We use the luminance measure.

2.2 Smoothing filter

This is used as a method of edge control. We want to avoid spurious high-frequency components from appearing as noise, while mostly preserving larger (true) edge features. A common choice is a (2D circular) Gaussian blur with a small blur radius. We use a Gaussian blur with standard deviation 2 for most of our experiments, although this depends on the density of the details in the image.

2.3 Finding the intensity gradients and directions

This is where the Sobel-Feldman (Sobel) filter comes in. The Sobel filters are small, separable 3×3 filters that provide us with integral approximations of the horizontal and vertical gradients of the intensity of the image.

$$T_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad T_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

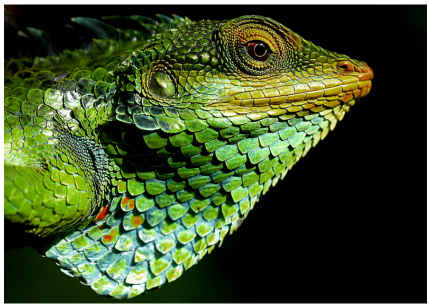
After finding the horizontal and vertical gradients, we find the (2D) image intensity gradient via the following formula:

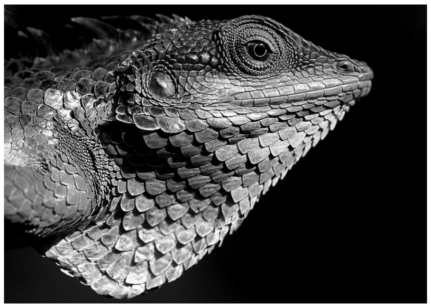
$$|G| = \sqrt{G_x^2 + G_y^2} \tag{1}$$

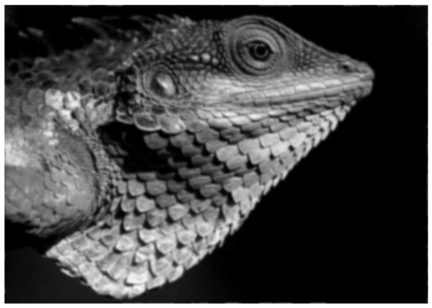
and we can find the direction of the gradient with:

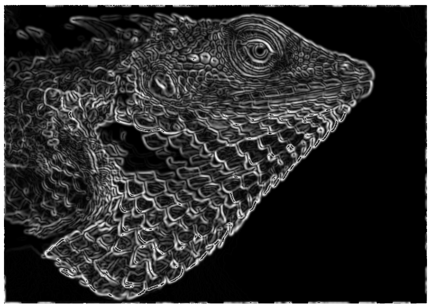
$$\angle G = \arctan \frac{G_y}{G_x} \tag{2}$$

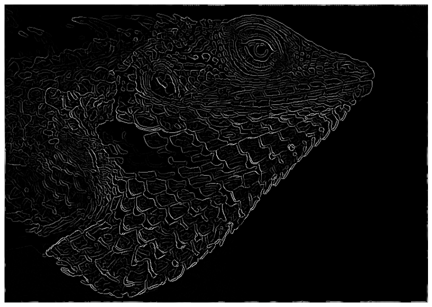
[2]`https://en.wikipedia.org/wiki/Canny_edge_detector`

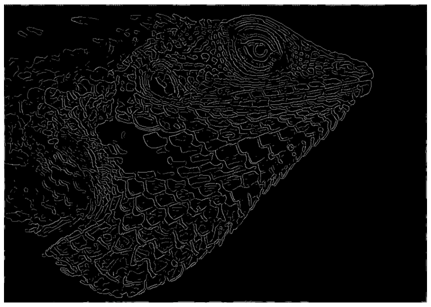

(a) Original image


(b) Luminance operator


(c) Gaussian blur


(d) Sobel filter


(e) Edge thinning


(f) Double thresholding

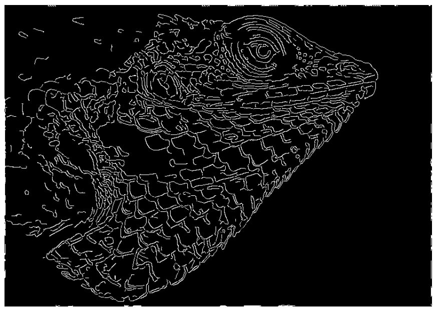
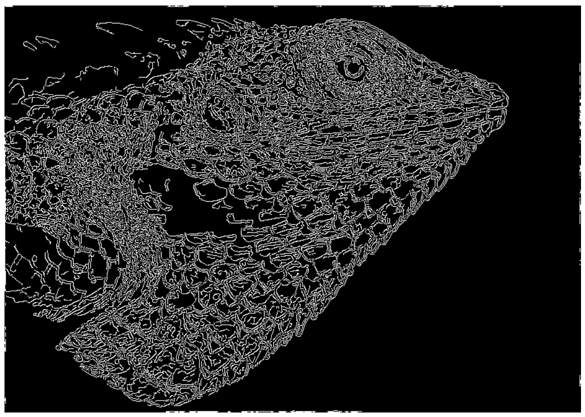
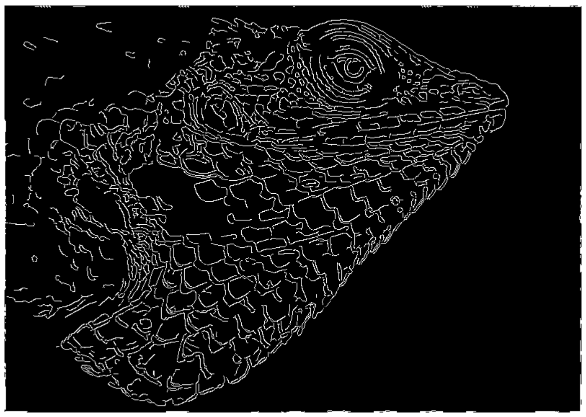
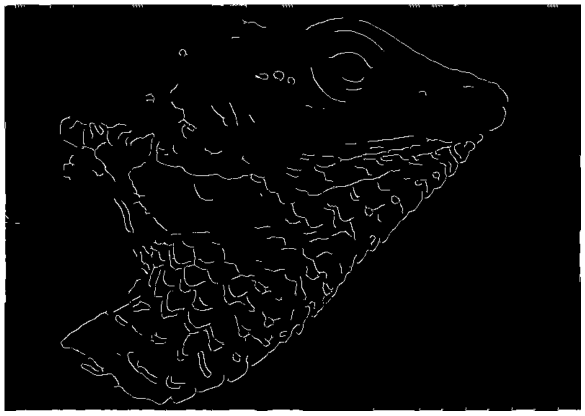

(g) Finished canny

Figure 1: Canny edge detection stages. Zoom in for detail. Babujayan, CC BY 3.0, via Wikimedia Commons.


(a) $\sigma = 1$


(b) $\sigma = 2$


(c) $\sigma = 3$

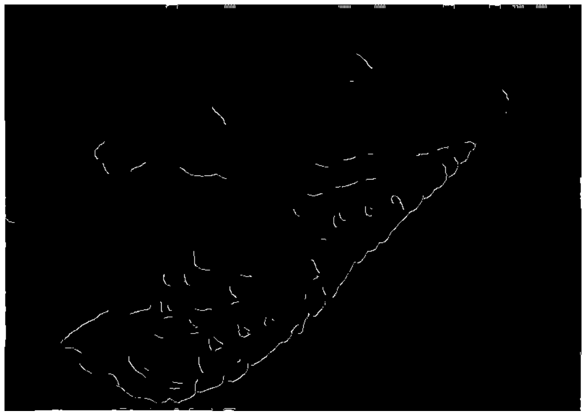

(d) $\sigma = 4$

Figure 2: Effect of changing blur standard deviation on the same lizard image. The blur size is dependent on the detail density (resolution) of the image. Increasing the blur size will decrease noise as well as the number of detected edges. It may also be necessary to change the thresholds based on the blur size. $\sigma = 2$ was chosen as the default for our tests as that seemed to work fairly well for reasonable resolutions.

2.4 Non-maximum edge suppression

The Sobel filter provides a fairly recognizable magnitude representation, but Canny decided that a more accurate representation of "edges" are represented by the zero-crossings of the second derivatives of the intensity gradients. This step uses the angle calculated in the previous step and determines if a pixel is a local maximum in of the directional intensity gradient (which corresponds to a zero-crossing in the second derivative). Usually, to simplify this calculation, the angle is quantized rounded into one of four angles $\{0, \pi/4, \pi/2, 3\pi/4\}$. (Note that opposite angles are not distinguishable, i.e., 0 and π are considered the same angle.) Values that are not local maxima are zeroed. The visual effect of this is that the edges become thinner.

2.5 Double-thresholding

This step and the next are further used to suppress noisy pixels. We mark remaining pixels above a high threshold as having a strong gradient, and mark pixels between that threshold and a lower threshold as having a weak gradient. Pixels with an intensity gradient lower than the lower threshold are zeroed.

2.6 Edge tracking via hysteresis

All weak pixels that are connected to a strong pixels are labeled strong. This process iteratively until there are no remaining weak pixels connected to a strong pixel. The end result of the Canny filter are the strong pixels, and the remaining weak pixels are zeroed. The intuition behind this step is that the strong pixels are already determined to be part of edges; the weak pixels, however, are more likely to come from noise, and are more likely to come from an edge if they are near other known edge pixels. This should be implemented via some sort of BFS algorithm.

3 Implementation and Optimizations

Each of the above steps roughly corresponds to one CUDA kernel. We will describe the naïve implementation and any optimizations performed for each step.

The CUDA program was implemented and tested using CUDA 9.2 on a GT 740, which has Compute Capability (CC) 3.0, on a Debian 10 computer with an i7-2600 CPU. The limiting factor of the GT 740 is that it has 1GB of VRAM, which caused out-of-memory (OOM) errors for images larger than 16k (15360×8640, or 132MP) – thus, our largest test cases are (roughly) 16k images.

3.1 Image preprocessing

Images were read and written using libpng. The code to process (read, decode, encode, write) the PNG files is courtesy of Guillaume Cottencau[3].

Libpng is (probably) not very efficient and the code is not very flexible, but it was simple enough for our needs.

3.2 Timing

A simple custom timer was written using the `clock()` function from the `<clock.h>` STL header. To prevent asynchronous kernel invocations, `cudaDeviceSynchronize()` was used when benchmarking the kernel elapsed times.

3.3 Grayscale

The grayscale operator is found in Figure 3. This uses the RGB-to-luminance formula from ITU BT.601[4].

$$I = 0.2989R + 0.5870G + 0.1140B \tag{3}$$

The first implementation used floating-point calculations, using this formula literally. Simply eliminating floating-point calculations as in Figure 3 reduced runtime by about 30%. To reduce time even further, we can use even lower precision, as color accuracy is not of utmost importance. Some other integral approximations can be found on the Internet[5].

3.4 Gaussian blurring

The convolution with the Gaussian blur is the slowest kernel. The original naive implementation involves a 2D kernel loaded into global memory and convolved at every pixel. The dimensions of the filter are determined by the blur standard deviation, and is calculated as so:

$$H = \lceil 6\sigma \rceil + 1$$

The (somewhat arbitrary) coefficient of 6 is to ensure that we capture enough of the Gaussian's filter; the +1 is used to make the filter an odd size to simplify calculations. Note that with a (somewhat reasonable) blur size $\sigma = 5$, this means that the filter is $31 \times 31 = 961$ pixels – this means 961 multiplications (and 1922 global memory accesses in the naive version with the filter in global memory) per thread.

The convolution then simply multiples pixels pointwise with the filter. There is a translation happening that causes the convolution to be centered around the image and not the filter (i.e., a "same" padding convolution). By moving

[3] `http://zarb.org/~gc/html/libpng.html`

[4] `https://stackoverflow.com/a/596241`

[5] `https://www.programmersought.com/article/20584662327/`

```
__global__ void toGrayScale(byte *dImg, byte *dImgMono, int h, int w, int ch)
{
        int ind, y, x;

        y = blockDim.y*blockIdx.y + threadIdx.y;
        x = blockDim.x*blockIdx.x + threadIdx.x;

        if (y >= h || x >= w) {
                return;
        }

        ind = y*w*ch + x*ch;

        dImgMono[y*w + x] = (2989*dImg[ind] + 5870*dImg[ind+1]
                + 1140*dImg[ind+2])/10000;
}

// convert back from single channel to multi-channel
__global__ void fromGrayScale(byte *dImgMono, byte *dImg, int h, int w, int ch)
{
        int ind, y, x;

        y = blockDim.y*blockIdx.y + threadIdx.y;
        x = blockDim.x*blockIdx.x + threadIdx.x;

        if (y >= h || x >= w) {
                return;
        }

        ind = y*w*ch + x*ch;
        dImg[ind] = dImg[ind+1] = dImg[ind+2] = dImgMono[y*w + x];
}
```

Figure 3: Color-to-grayscale and grayscale-to-3 channel kernels.

```
__global__ void conv2d(byte *d1, byte *d3,
        int h1, int w1, int h2, int w2)
{
        int y, x, i, j, imin, imax, jmin, jmax, ip, jp;
        float sum;

        // infer y, x, from block/thread index
        y = blockDim.y * blockIdx.y + threadIdx.y;
        x = blockDim.x * blockIdx.x + threadIdx.x;

        // out of bounds, no work to do
        if (x >= w1 || y >= h1) {
                return;
        }

        // appropriate ranges for convolution
        imin = max(0, y+h2/2-h2+1);
        imax = min(h1, y+h2/2+1);
        jmin = max(0, x+w2/2-w2+1);
        jmax = min(w1, x+w2/2+1);

        // convolution
        sum = 0;
        for (i = imin; i < imax; ++i) {
                for (j = jmin; j < jmax; ++j) {
                        ip = i - h2/2;
                        jp = j - w2/2;

                        sum += d1[i*w1 + j] * dFlt[(y-ip)*w2 + (x-jp)];
                }
        }

        // set result
        d3[y*w1 + x] = sum;
}
```

Figure 4: Naive convolution kernel

the kernel into constant memory, we get the (still naive) implementation found in Figure 4. This simple optimization halved the runtime of the convolution kernel, probably due to the fact that there were half as many global memory accesses.

The first optimization we can make is to make the convolution a “tiled” memory algorithm, in which we preload a section of the image into the block shared memory so that we reduce the number of global memory accesses. However, we have to be careful to load all of the pixels necessary for the convolution: the convolution for one block of pixels requires a surrounding “apron” of adjacent pixels, as shown in Figure 5a. This means that each thread loads one pixel, and some threads on the border of the thread block will be loading pixels but not performing a filter calculation; or, conversely, each thread performs a convolution operation but threads on the border of the thread block also load adjacent apron pixels into shared memory. Either approach is fine, but we chose the former for its simplicity. This means that in each thread block, some threads are dedicated only to loading apron pixels. Now, each thread only performs one global memory access; however, we need more threads (more blocks) to process the image because not all threads are used anymore for calculating a convolution. Each pixel is loaded from global memory up to nine times, a huge improvement for larger kernels.

Note that having a large apron is wasteful; each apron pixel is loaded into shared memory several times and the corresponding threads are inactive during the filter computation. Thus, if the apron (filter) size is large relative to the block size, this becomes increasingly inefficient. In the example shown in Figure 5b, only one-ninth of the pixels actually perform filter calculations, and each pixel gets loaded from global memory exactly nine times. This is similar to the case of a 31×31 Gaussian filter in a 32×32 thread block.

The solution to our woes is the fact that a 2D Gaussian convolution is separable – we can decompose it into a vertical (1D) convolution followed by a horizontal (1D) convolution. The reason for the efficiency is two-fold: firstly, we only have to perform $2H$ rather than H^2 mult-adds, because our filter is linear (with the same length as the side-length of the 2D filter). Secondly, we only need to worry about loading apron pixels along the convolution axis. Each apron pixel gets loaded exactly twice from global memory. Furthermore, if we elongate the block so that its dimension along the convolution axis is longer than in the perpendicular direction, we can minimize the number of apron pixels loaded into memory; this is shown in Figure 5c.

These principles were applied to result in the 1-D convolution kernel shown in Figure 6. Only the horizontal filter is shown, but the same concept is applied to create a vertical filter, and the two are performed in serial. For the horizontal convolution, the block size is set to 16×64; for the vertical convolution, the block size is set to 64×16.

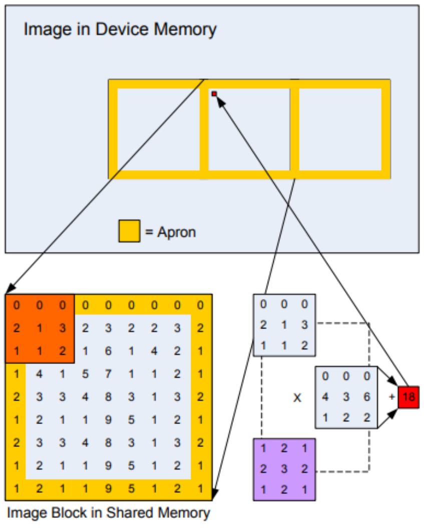
3.5 Sobel filter

The Sobel filter is simply a 2-D convolution, but it is a small fixed size (3×3). It is also separable into two linear filters (3×1 and 1×3), which reduces the number of computations. Given the small filter size and the few global memory accesses, we found that the optimizations did not help by much.

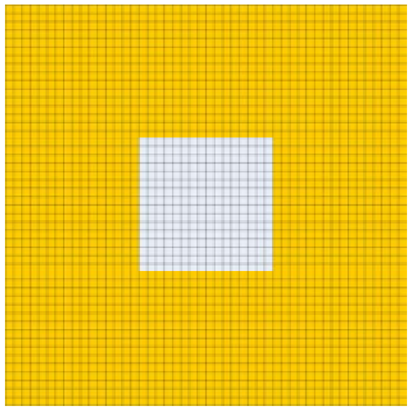
The naive version, shown in Figure 7, performed only slightly worse than the version using shared memory and separable filters, shown in Figure 8. A version using shared memory but without separating the filter, shown in Figure 9, performs slightly better than the separable version.

3.6 Non-maximum edge suppression and double-thresholding

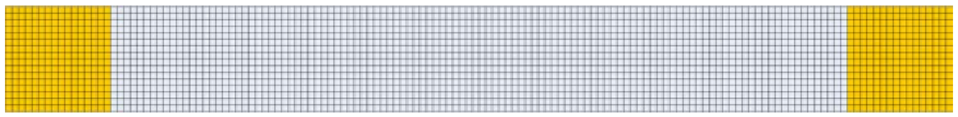
The implementation for these two sections are fairly straightforward and can't be optimized much. They each require few global memory accesses, do not



(a) Visual of the convolution apron



(b) An large (inefficient) apron



(c) Efficiency with a 1D convolution

Figure 5: Considerations with the convolution apron. Images source: [3].

```
__global__ void conv1dRows(byte *dIn, byte *dOut, int h, int w, int fltSize)
{
        int y, x, as, i, j;
        float sum;
        __shared__ byte tmp[lbs*sbs];

        as = fltSize>>1; // apron size

        // infer y, x, from block/thread index
        // note extra operations based on apron for x
        y = sbs * blockIdx.y + ty;
        x = (lbs-(as<<1)) * blockIdx.x + tx-as;

        // load data
        if (y<h && x>=0 && x<w) {
                tmp[ty*lbs+tx] = dIn[y*w+x];
        }

        __syncthreads();

        // perform 1-D convolution
        if (tx>=as && tx<lbs-as && y<h && x<w) {
                for (i = ty*lbs+tx-as, j = 0, sum = 0; j < fltSize; ++i, ++j) {
                        sum += dFlt[j] * tmp[i];
                }

                // set result
                dOut[y*w+x] = sum;
        }
}
```

Figure 6: Efficient 1-D convolution kernel

```
__global__ void sobel(byte *img, byte *out, byte *out2, int h, int w)
{
        int vKer, hKer, y, x;

        y = blockDim.y*blockIdx.y + threadIdx.y;
        x = blockDim.x*blockIdx.x + threadIdx.x;

        // make sure not on edge
        if (y <= 0 || y >= h-1 || x <= 0 || x >= w-1) {
                return;
        }

        vKer = img[(y-1)*w+(x-1)]*1 + img[(y-1)*w+x]*2 + img[(y-1)*w+(x+1)]*1 +
                img[(y+1)*w+(x-1)]*-1 + img[(y+1)*w+x]*-2 + img[(y+1)*w+(x+1)]*-1;

        hKer = img[(y-1)*w+(x-1)]*1 + img[(y-1)*w+(x+1)]*-1 +
                img[y*w+(x-1)]*2 + img[y*w+(x+1)]*-2 +
                img[(y+1)*w+(x-1)]*1 + img[(y+1)*w+(x+1)]*-1;

        out[y*w+x] = out[y*w+x] = sqrtf(hKer*hKer + vKer*vKer);
        out2[y*w+x] = (byte)((atan2f(vKer,hKer)+9/8*M_PI)*4/M_PI)&0x3;
}
```

Figure 7: Naïve Sobel convolution

cause any major branch diversions, and as a result take the least time of all the kernels. See Figures 10 and 11 for the implementations.

3.7 Edge tracking via hysteresis

This method closely mirrors that of Luo and Duraiswami [1]. The general idea is to emulate a BFS using tiled memory. What makes it difficult, as Luo and Duraiswami point out, is that this is a nonlocal problem; unlike any of the other kernels so far, one "strong" edge may affect a "weak" edge at an arbitrary distance connected by some path. We follow the heuristic multi-pass approach given by Luo and Duraiswami. The general idea is to introduce an apron of width 1 so that a block can "discover" strong edges from adjacent blocks. In each pass, the block performs a local iterative BFS, updating weak edges connected to strong edges until there are no more remaining. After each pass, the results are written back to global memory so they can be retrieved by adjacent blocks in the next kernel invocation. A visual of the algorithm is shown in Figure 12.

Like Luo and Duraiswami, we choose an arbitrary number of hysteresis passes to perform (we perform five iterations). The effect of increasing the number of passes is shown in Figure 13.

The abundant `__syncthreads()` usage is probably not optimal, but it was a necessary workaround to prevent race conditions.

The initial implementation using global memory is shown in Figure 14. The

```
__global__ void sobel_sep(byte *img, byte *out, byte *out2, int h, int w)
{
        int y, x;

        // using int instead of byte for the following offers a 0.01s (5%)
        // speedup on the 16k image -- coalesced memory?
        int vKer, hKer;
        __shared__ int tmp1[bs*bs], tmp2[bs*bs], tmp3[bs*bs];

        y = (bs-2)*blockIdx.y + threadIdx.y-1;
        x = (bs-2)*blockIdx.x + threadIdx.x-1;

        // load data from image
        if (y>=0 && y<h && x>=0 && x<w) {
                tmp1[ty*bs+tx] = img[y*w+x];
        }

        __syncthreads();

        // first convolution
        if (ty>=1 && ty<bs-1 && tx && tx<bs) {
                tmp2[ty*bs+tx] = tmp1[(ty-1)*bs+tx]
                        + (tmp1[ty*bs+tx]<<1) + tmp1[(ty+1)*bs+tx];
        }

        if (ty && ty<bs && tx>=1 && tx<bs-1) {
                tmp3[ty*bs+tx] = tmp1[ty*bs+(tx-1)]
                        + (tmp1[ty*bs+tx]<<1) + tmp1[ty*bs+(tx+1)];
        }

        __syncthreads();

        // second convolution and write-back
        if (ty>=1 && ty<bs-1 && tx>=1 && tx<bs-1 && y<h && x<w) {
                hKer = tmp2[ty*bs+(tx-1)] - tmp2[ty*bs+(tx+1)];
                vKer = tmp3[(ty-1)*bs+tx] - tmp3[(ty+1)*bs+tx];

                out[y*w+x] = sqrtf(hKer*hKer + vKer*vKer);
                out2[y*w+x] = (byte)((atan2f(vKer,hKer)+9/8*M_PI)*4/M_PI)&0x3;
        }
}
```

Figure 8: Sobel convolution using shared memory and separable filters

```
__global__ void sobel_shm(byte *img, byte *out, byte *out2, int h, int w)
{
        int y, x;
        int vKer, hKer;
        __shared__ int tmp[bs*bs];

        y = (bs-2)*blockIdx.y + threadIdx.y-1;
        x = (bs-2)*blockIdx.x + threadIdx.x-1;

        // load data from image
        if (y>=0 && y<h && x>=0 && x<w) {
                tmp[ty*bs+tx] = img[y*w+x];
        }

        __syncthreads();

        // convolution and write-back
        if (ty>=1 && ty<bs-1 && tx>=1 && tx<bs-1 && y<h && x<w) {
                vKer = tmp[(ty-1)*bs+(tx-1)]*1 + tmp[(ty-1)*bs+tx]*2
                        + tmp[(ty-1)*bs+(tx+1)]*1 + tmp[(ty+1)*bs+(tx-1)]*-1
                        + tmp[(ty+1)*bs+tx]*-2 + tmp[(ty+1)*bs+(tx+1)]*-1;

                hKer = tmp[(ty-1)*bs+(tx-1)]*1 + tmp[(ty-1)*bs+(tx+1)]*-1 +
                       tmp[ty*bs+(tx-1)]*2 + tmp[ty*bs+(tx+1)]*-2 +
                       tmp[(ty+1)*bs+(tx-1)]*1 + tmp[(ty+1)*bs+(tx+1)]*-1;

                out[y*w+x] = sqrtf(hKer*hKer + vKer*vKer);
                out2[y*w+x] = (byte)((atan2f(vKer,hKer)+9/8*M_PI)*4/M_PI)&0x3;
        }
}
```

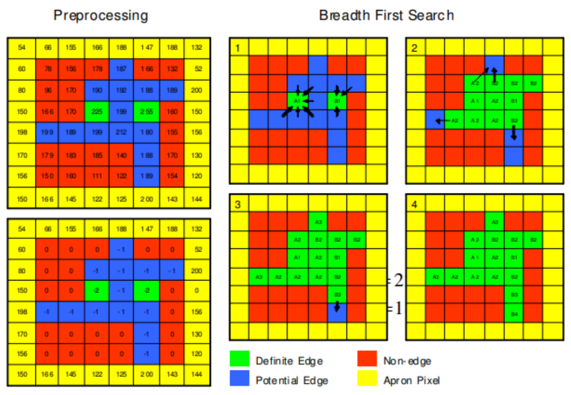
Figure 9: Sobel convolution using shared memory and 2D filter

```
__global__ void edge_thin(byte *mag, byte *angle, byte *out, int h, int w)
{
        int y, x, y1, x1, y2, x2;

        y = blockDim.y*blockIdx.y + threadIdx.y;
        x = blockDim.x*blockIdx.x + threadIdx.x;

        // make sure not on the border
        if (y <= 0 || y >= h-1 || x <= 0 || x >= w-1) {
                return;
        }

        // if not greater than angles in both directions, then zero
        switch (angle[y*w + x]) {
        case 0:
                // horizontal
                y1 = y2 = y;
                x1 = x-1;
                x2 = x+1;
                break;
        case 3:
                // 135
                y1 = y-1;
                x1 = x+1;
                y2 = y+1;
                x2 = x-1;
                break;
        case 2:
                // vertical
                x1 = x2 = x;
                y1 = y-1;
                y2 = y+1;
                break;
        case 1:
                // 45
                y1 = y-1;
                x1 = x-1;
                y2 = y+1;
                x2 = x+1;
        }

        if (mag[y1*w + x1] >= mag[y*w + x] || mag[y2*w + x2] >= mag[y*w + x]) {
                out[y*w + x] = 0;
        } else {
                out[y*w + x] = mag[y*w + x];
        }
}
```

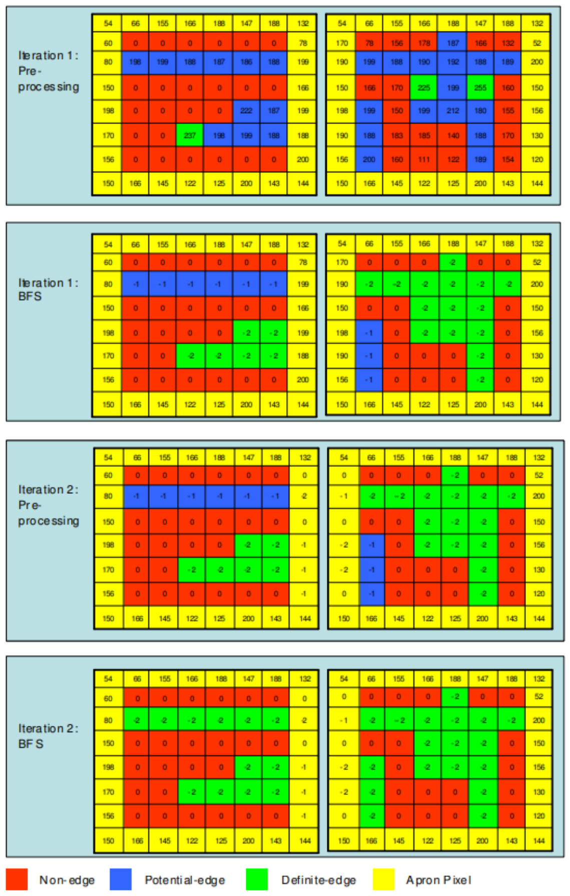
Figure 10: Edge thinning kernel

```
#define MSK_LOW         0x0     // below threshold 1
#define MSK_THR         0x60    // at threshold 1
#define MSK_NEW         0x90    // at threshold 2, newly discovered
#define MSK_DEF         0xff    // at threshold 2 and already discovered

// perform double thresholding
__global__ void edge_thin(byte *dImg, byte *out, int h, int w, byte t1, byte t2)
{
        int y, x, ind, grad;

        y = blockDim.y*blockIdx.y + threadIdx.y;
        x = blockDim.x*blockIdx.x + threadIdx.x;

        if (y >= h || x >= w) {
                return;
        }

        ind = y*w + x;
        grad = dImg[ind];
        if (grad < t1) {
                out[ind] = MSK_LOW;
        } else if (grad < t2) {
                out[ind] = MSK_THR;
        } else {
                out[ind] = MSK_NEW;
        }
}
```

Figure 11: Double-thresholding kernel

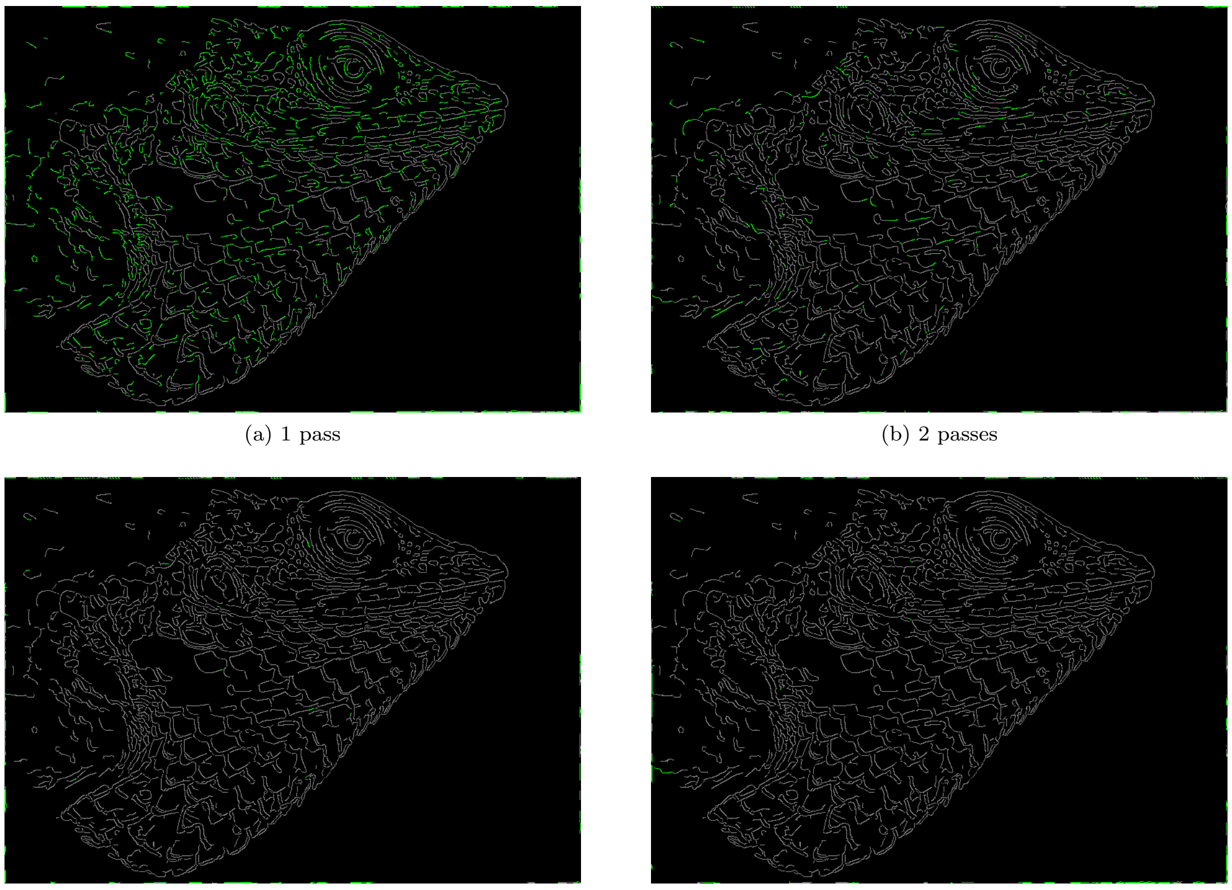


(a) Hysteresis edge tracking within a block (BFS in shared memory)



(b) Edge tracking across blocks (multiple passes)

Figure 12: Hysteresis algorithm. Images source: [1].


(a) 1 pass

(b) 2 passes

(c) 3 passes

(d) 4 passes

Figure 13: Effect of varying the number of hysteresis passes on the lizard image. The green pixels indicate new strong edge pixels found in the current hysteresis pass, and gray pixels indicate edges that were already marked strong before the current pass. Very few edges are added after the third iteration. We chose five iterations as the default for our tests.

tiled version is shown in Figure 15.

3.8 CPU implementation

The CPU implementation follows the same logic as the CUDA naive versions. We cannot make some of the optimizations (e.g., shared memory – the CPU cache is not large enough). Most of the algorithms were implemented on a pixel-by-pixel basis by simply looping through every pixel with a double loop.

The main differences lie in the setup and the hysteresis algorithm. The CPU version does not require copying the image buffer from host to device.

The hysteresis version is not performed in parallel passes like the CUDA version for obvious reasons. Instead, a DFS is performed on each pixel. Since this involves nonlocal memory that requires expensive synchronization between blocks (i.e., writing to) on the CUDA version – this makes the speedup on the hysteresis section the smallest of all of the kernels.

The only optimization that was performed on the GPU that could also be performed on the CPU is the separable blurring filter, but that was not implemented. This is left for future investigation.

4 Results

For testing, we mostly focused on four large images downloaded from the image, which will henceforth be referred to as "lizard"[6], "rocks"[7], "moon"[8], and "skyline14k"[9]. Several scaled versions of the skyline image were also used.

As expected, the grayscale, edge thinning, and thresholding operators are the quickest. These all involve a single linear pass over the data and only a few memory accesses per pixel. The sobel and blurring operators take the most time. These results can be seen in Tables 1 and 2. The mean time reduction for the edge thinning and thresholding from the CPU to CUDA versions is 93% and 86%, respectively.

The hysteresis operator is one of the faster operators for the CPU version, while it is the slowest operator (counting overall time for five iterations of the kernel) on the CUDA version. This is as expected: on the CPU, we have much faster control flow operators and nonlocal caching using a DFS, However, in CUDA, since each warp has to perform the same instruction, when any path is being traced all of the other threads in the warp must remain idle until all paths are traced. Additionally, we need multiple global memory write-backs because edges may pass between thread block tile boundaries. As a result, the CPU hysteresis is actually faster than the CUDA hysteresis for the smallest test image (lizard), and on the same order of magnitude for the other three test images.

[6]Babujayan, CC BY 3.0, via Wikimedia Commons.

[7]`https://wallpapercave.com/w/wp1848524`

[8]`https://www.reddit.com/r/pics/comments/dszr27`

[9]Kristoffer Trolle from Copenhagen, Denmark, CC BY 2.0, via Wikimedia Commons.

```
// check and set neighbor
#define CAS(buf, cond, x2, y2, width) \
        if ((cond) && (buf)[(y2)*(width)+(x2)] == MSK_THR) \
                (buf)[(y2)*(width)+(x2)] = MSK_NEW

// perform one iteration of hysteresis
__global__ void hysteresis(byte *dImg, int h, int w, bool final)
{
        int y, x;
        __shared__ byte changes;

        // infer y, x, from block/thread index
        y = blockDim.y * blockIdx.y + threadIdx.y;
        x = blockDim.x * blockIdx.x + threadIdx.x;

        // check if pixel is connected to its neighbors; continue until
        // no changes remaining
        do {
                __syncthreads();
                changes = 0;
                __syncthreads();

                // make sure inside bounds -- need this here b/c we can't have
                // __syncthreads() cause a branch divergence in a warp;
                // see https://stackoverflow.com/a/6667067/2397327

                // if newly-discovered edge, then check its neighbors
                if ((x<w && y<h) && dImg[y*w+x] == MSK_NEW) {
                        // promote to definitely discovered
                        dImg[y*w+x] = MSK_DEF;
                        changes = 1;

                        // check neighbors
                        CAS(dImg, x>0&&y>0,     x-1, y-1, w);
                        CAS(dImg, y>0,          x,   y-1, w);
                        CAS(dImg, x<w-1&&y>0,   x+1, y-1, w);
                        CAS(dImg, x<w-1,        x+1, y,   w);
                        CAS(dImg, x<w-1&&y<h-1, x+1, y+1, w);
                        CAS(dImg, y<h-1,        x,   y+1, w);
                        CAS(dImg, x>0&&y<h-1,   x-1, y+1, w);
                        CAS(dImg, x>0,          x-1, y,   w);
                }

                __syncthreads();
        } while (changes);

        // set all threshold1 values to 0
        if (final && (x<w && y<h) && dImg[y*w+x] != MSK_DEF) {
                dImg[y*w+x] = 0;
        }
}
```

Figure 14: Naïve hysteresis using global memory

```
__global__ void hysteresis_shm(byte *dImg, int h, int w, bool final)
{
        int y, x;
        bool in_bounds;
        __shared__ byte changes, tmp[bs*bs];

        // infer y, x, from block/thread index
        y = (bs-2)*blockIdx.y + ty-1;
        x = (bs-2)*blockIdx.x + tx-1;

        in_bounds = (x<w && y<h) && (tx>=1 && tx<bs-1 && ty>=1 && ty<bs-1);

        if (y>=0 && y<h && x>=0 && x<w) {
                tmp[ty*bs+tx] = dImg[y*w+x];
        }

        __syncthreads();

        // check if pixel is connected to its neighbors; continue until
        // no changes remaining
        do {
                __syncthreads();
                changes = 0;
                __syncthreads();

                // if newly-discovered edge, then check its neighbors
                if (in_bounds && tmp[ty*bs+tx] == MSK_NEW) {
                        // promote to definitely discovered
                        tmp[ty*bs+tx] = MSK_DEF;
                        changes = 1;

                        // check neighbors
                        CAS(tmp, 1,            tx-1, ty-1, bs);
                        CAS(tmp, 1,            tx,   ty-1, bs);
                        CAS(tmp, x<w-1,        tx+1, ty-1, bs);
                        CAS(tmp, x<w-1,        tx+1, ty,   bs);
                        CAS(tmp, x<w-1&&y<h-1, tx+1, ty+1, bs);
                        CAS(tmp, y<h-1,        tx,   ty+1, bs);
                        CAS(tmp, y<h-1,        tx-1, ty+1, bs);
                        CAS(tmp, 1,            tx-1, ty,   bs);
                }

                __syncthreads();
        } while (changes);

        if (y>=0 && y<h && x>=0 && x<w) {
                if (final) {
                        if (in_bounds) {
                                dImg[y*w+x] = MSK_DEF*(tmp[ty*bs+tx]==MSK_DEF);
                        }
                } else {
                        dImg[y*w+x] = max(dImg[y*w+x], tmp[ty*bs+tx]);
                }
        }
}
```

Figure 15: Hysteresis using shared memory

We were able to achieve minor speedups with each of the optimizations. The previously discussed results are the most optimized versions. Table 3 lists the times of the less-optimized versions (except for the unoptimized blur kernel, which will be discussed separately.) Switching the grayscale operator from using floating-point operations to integral ones caused a 26% decrease in kernel time. Switching hysteresis to use shared memory caused a 52% speedup (this is displayed visually in Figure 16a).

The speedup from the Sobel filter optimizations was milder, likely due to the fact that it is already a relatively simple operator. Using shared memory (tiling) with a separable filter is actually a little slower than using a non-separable filter, most likely because of requiring two thread barriers rather than a single one. The separable version offers a 7% time reduction, and the non-separable version offers a 10% reduction over the naive implementation. See Figure 16b.

Lastly, an additional "optimization" to remove all `cudaDeviceSynchronize()` calls was made to see the effect of these synchronization barriers. There is a consistent but very small effect (on the order of thousandths of seconds for each test image).

In Tables 4 and 5, we see that the times scale very linearly with the number of pixels in the image. (This is true for all but the CPU blur filter, which will be discussed in the following paragraph.) This is shown visually in Figure 16d.

The largest speedup by far was achieved in the blur filter, not least because it involves the most computations and is the slowest kernel in general. Table 6 and Figure 16c demonstrates the quadratic growth of the non-separable version as the blur size increases, as opposed to the linear growth with the separable version. For most of our tests, the blur size had $\sigma = 2$, at which the time spent is reduced on average by 82%. For $\sigma = 5$, the reduction is 93%. (The (normalized) time reduction is unbelievably consistent, varying by less than 1% – what appears to be two lines in Figure 16c is actually 8 lines, with each of the four images represented.)

The CPU version implements the naive blur and thus is also clearly quadratic. For $\sigma = 2$ ($H = 13$, which means 169 memory accesses and mult-adds per pixel), the naive CUDA version (using the same algorithm) offers a 99.3% time reduction, and thus the optimized CUDA version offers a 99.98% speedup. Since the overall time (for the CPU version) is dominated by the blurring time, the overall time reduction for the images is also over 99%.

In Table 7, the images produced by the CPU and CUDA versions are compared. The resulting accuracy metric is simply the percentage of pixels that match in the final image. Some of the smaller images report a larger error margin; this is probably due to the fact that the boundary pixels are not properly accounted for in either of the latest implementations during the convolutions. This should be fixed for a future comparison. Much of the remaining differences should result from hysteresis, as this is the only non-deterministic algorithm (in the CUDA version only, as it depends on which warps get scheduled to run first). However, the numbers are generally high enough to indicate that most pixels were the same; the mean accuracy over the test images is roughly 99%.

image	lizard	rocks	moon	skyline14k
width	4444	7680	14694	14091
height	3136	4320	8266	9394
pixels (MP)	13.94	33.1776	121.46	132.37
blur	16.392047	38.991000	144.943194	158.283468
sobel	0.503448	1.370735	4.346238	5.109478
edgethin	0.127931	0.392264	1.259965	1.390145
threshold	0.044430	0.096321	0.345455	0.407205
hysteresis	0.039386	0.154966	0.443695	0.559849
overall	17.294973	41.319512	152.664951	167.053442

Table 1: CPU timings for processing the same images as in Table 2.

image	lizard	rocks	moon	skyline14k
gray	0.007841	0.017504	0.068982	0.073921
blur (separable)	0.024333	0.057933	0.207798	0.225898
sobel (shared mem)	0.019585	0.052028	0.201424	0.203151
edgethin	0.009991	0.024648	0.090547	0.098592
threshold	0.005778	0.013698	0.049114	0.054493
hysteresis (shared mem)	0.008176	0.017076	0.058182	0.068109
hyst total	0.040881	0.085383	0.290910	0.340547
overall	0.117525	0.270621	0.983041	1.076650
nosync	0.116558	0.267512	0.979548	1.071600

Table 2: GPU kernel and overall timings for optimized kernels on four test images. For these tests, the following hyperparameters were used: blur $\sigma = 2$, threshold 1=0.2, threshold 2=0.4, and 5 hysteresis passes. Additionally, the nosync row indicates the overall time if no `cudaDeviceSynchronize()` calls were made; the effect of `cudaDeviceSynchronize()` is fairly negligible.

image	lizard	rocks	moon	skyline14k
gray (fp)	0.010738	0.023893	0.092579	0.100435
sobel (naive)	0.022106	0.056393	0.225306	0.222554
sobel (separable)	0.020585	0.052922	0.206515	0.210145
hysteresis (naive)	0.016735	0.034941	0.120156	0.139790
hyst total	0.083675	0.174705	0.600781	0.698950

Table 3: GPU kernel timings for unoptimized kernels on the same four test images. The same canny parameters were used.

image	skyline500	skyline1000	skyline2.5k	skyline5k	skyline10k	skyline14k
width	500	1000	2500	5000	10000	14091
height	333	667	1667	3334	6667	9394
pixels (MP)	0.17	0.67	4.17	16.67	66.67	132.37
blur	0.193700	0.779473	4.867015	19.478580	79.527461	158.283468
sobel	0.006869	0.027894	0.149842	0.584700	2.346041	5.109478
edgethin	0.001850	0.007012	0.039814	0.154795	0.613782	1.390145
threshold	0.000578	0.002514	0.012705	0.048452	0.189971	0.407205
hysteresis	0.001592	0.006575	0.022750	0.079599	0.293580	0.559849
overall	0.206662	0.830951	5.131639	20.503200	83.607380	167.053442

Table 4: CPU timings for processing the same images as in Table 5.

image	skyline500	skyline1k	skyline2.5k	skyline5k	skyline10k	skyline14k
gray	0.000265	0.000737	0.002767	0.009557	0.035608	0.073921
blur	0.000756	0.001451	0.007824	0.029554	0.113994	0.225898
sobel	0.000299	0.001084	0.006579	0.025391	0.101429	0.203151
edgethin	0.000385	0.000772	0.003750	0.012856	0.049072	0.098592
threshold	0.000321	0.000521	0.001737	0.007436	0.027187	0.054493
hysteresis	0.000243	0.000551	0.002589	0.009209	0.034696	0.068109
hysteresis total	0.001217	0.002757	0.012947	0.046046	0.173481	0.340547
overall	0.003606	0.008689	0.039141	0.141744	0.539666	1.076650

Table 5: GPU kernel timings for optimized kernels on the same image at six resolutions.

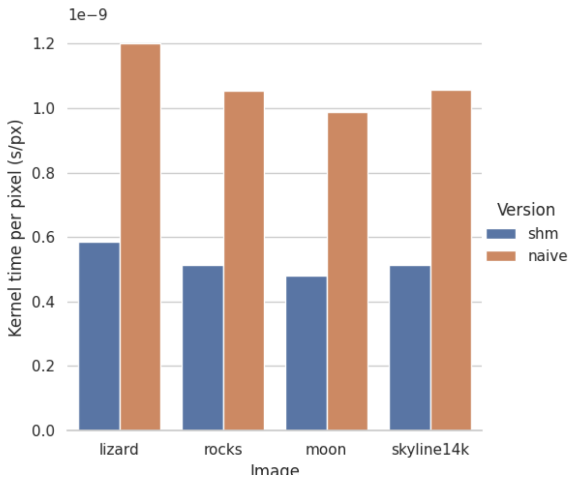
image	lizard	rocks	moon	skyline14k
naive $\sigma = 1$	0.048326	0.115943	0.421206	0.456401
naive $\sigma = 2$	0.132040	0.313778	1.153320	1.254950
naive $\sigma = 3$	0.277972	0.658977	2.413430	2.624620
naive $\sigma = 4$	0.473423	1.119180	4.125480	4.493010
naive $\sigma = 4$	0.707206	1.691080	6.175220	6.761020
separable $\sigma = 1$	0.018326	0.042776	0.157439	0.169571
separable $\sigma = 2$	0.024333	0.057933	0.207798	0.225898
separable $\sigma = 3$	0.029375	0.068700	0.249971	0.274145
separable $\sigma = 4$	0.037768	0.088535	0.325202	0.355358
separable $\sigma = 5$	0.049688	0.117644	0.426353	0.465503

Table 6: Comparison of timings for the naive and optimized Gaussian convolutions on the four test images for different blur standard deviations.

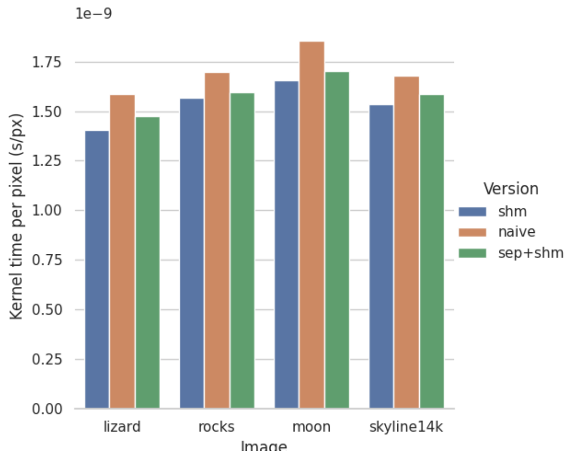
image	lizard	moon	rocks	skyline14k
accuracy	0.9767	0.994778	0.988704	0.994285

image	skyline500	skyline1k	skyline2.5k	skyline5k	skyline10k
accuracy	0.968613	0.978121	0.985476	0.990231	0.992143

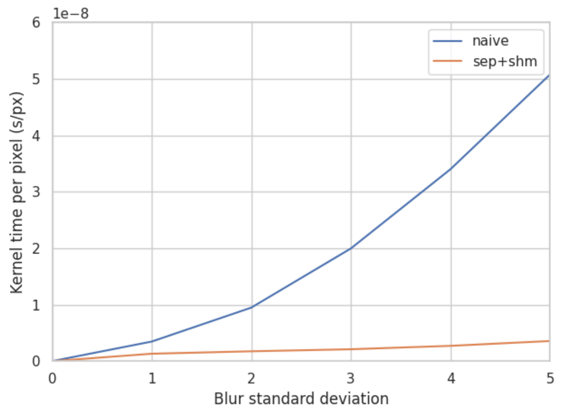
Table 7: Percent of pixels matching from CPU and GPU versions



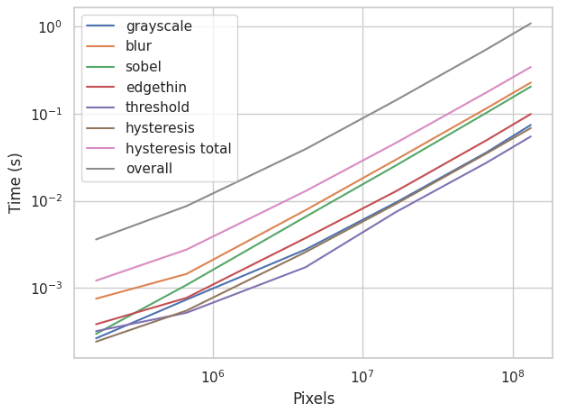
(a) Comparison of normalized kernel times of the two hysteresis implementations.



(b) Comparison of normalized kernel times of the three Sobel filter implementations. The two optimized version perform very similarly, but the non-separable one probably wins out due to having fewer synchronization barriers.



(c) Comparison of the normalized kernel times of the two (Gaussian) convolution implementations, shown over several standard deviations $\sigma = \{0, 1, 2, 3, 4, 5\}$. Not only does the separable filter beat the 2D filter by a large margin, but it scales linearly with blur size rather than quadratically.



(d) Timings of the different kernels on images of different sizes. All of the kernel timings grow linearly w.r.t. the number of pixels in the image, as expected. This is performed on six different resolutions of the cityscape image with the same Canny parameters.

Figure 16: Result of optimizing kernels, and overall scalability. (a), (b), and (c) are normalized by the number of pixels in the image. (d) is shown on a log-log scale and demonstrates that all of the algorithms scale linearly with number of pixels (this allows us to normalize by pixels count and get similar results).

5 Conclusion

We were able to achieve a 99.4% speedup using optimized CUDA C++ over a naive implementation on the CPU written in C on four large test images. Simply by converting what would a pixel-by-pixel algorithm to a parallel kernel offered over 90% time reduction for all kernels except hysteresis. Most of the time reduction came from the blurring kernel; the naive version achieved a 99.3% time reduction and the optimized version caused a 99.98% time reduction. As expected, the hysteresis kernel runtime is comparable to that of the CPU version, because of the nonlocal memory accesses and complex nonuniform branching requirements.

For future research, we may attempt separating the blur kernel for the CPU in the same manner as for CUDA. Further optimizations for the CUDA version may include optimizing for the capabilities of the device (our target was the GT 740, but its capabilities may be different from newer NVIDIA processors) and accounting for memory coalescence by properly aligning tiles. It may also be a good idea to test this against comparable versions written in OpenCV/OpenCL and MATLAB; Luo and Duraiswami [1] perform tests against OpenCV and MATLAB, but the hardware capabilities and software improvements (e.g., the introduction of OpenCL) since then have greatly changed.

References

[1] Yuancheng Luo and Ramani Duraiswami. Canny edge detection on nvidia cuda. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8. IEEE, 2008.

[2] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[3] Victor Podlozhnyuk. Image convolution with cuda. *NVIDIA Corporation white paper, June*, 2097(3), 2007.