Canny Filter Implementation in CUDA C++

ECE453 Final Project

Jonathan Lam

The Cooper Union for the Advancement of Science and Art

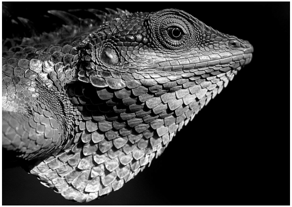
May 10, 2021

Objective:

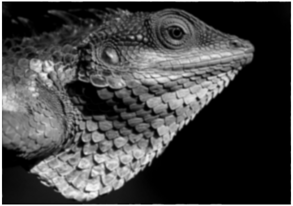
- Implement Canny edge detection algorithm in CUDA C++ and CPU (C/C++)
- Compare performance differences and accuracy
- Attempt further optimizations in CUDA

Work is based on Luo and Duraiswami [2].

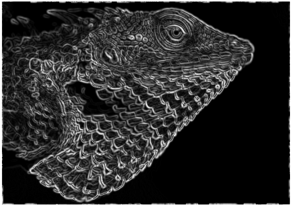
(a) Original image



(b) Luminance operator

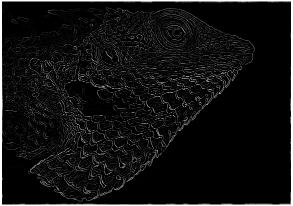


(c) Gaussian blur

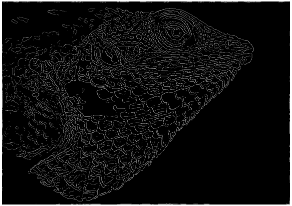


(d) Sobel filter

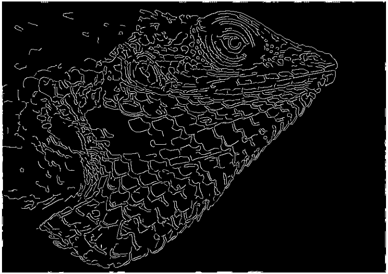
Figure: Canny edge detection stages. Zoom in for detail. Babujayan, CC BY 3.0, via Wikimedia Commons.



(e) Edge thinning

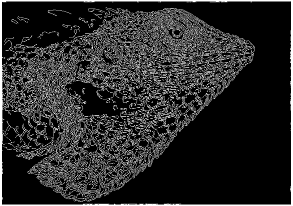


(f) Double thresholding

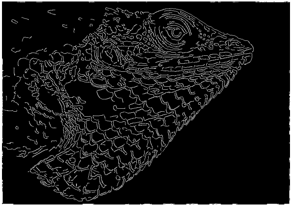


(g) Finished canny

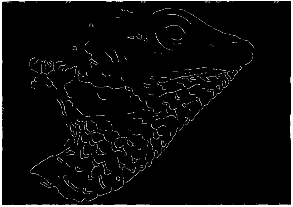
Figure: Canny edge detection stages. Zoom in for detail. Babujayan, CC BY 3.0, via Wikimedia Commons.



(a) $\sigma = 1$



(b) $\sigma = 2$



(c) $\sigma = 3$

(d) $\sigma = 4$

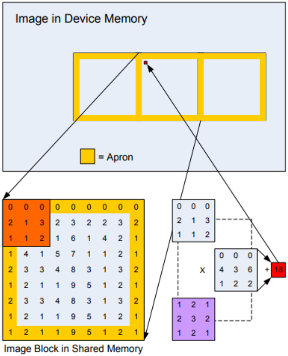
Figure: (cont'd.) Effect of changing blur standard deviation on the same lizard image. The blur size is dependent on the detail density (resolution) of the image. Increasing the blur size will decrease noise as well as the number of detected edges. It may also be necessary to change the thresholds based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

```
__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];
}
```

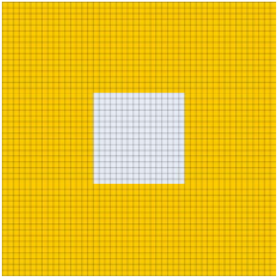
Figure: Color-to-grayscale and grayscale-to-3 channel kernels.

```
__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;
}
```

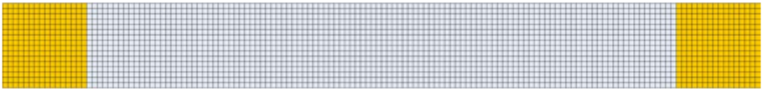
Figure: Naive convolution kernel



(a) Visual of the convolution apron



(b) An large (inefficient) apron



(c) Efficiency with a 1D convolution

Figure: Considerations with the convolution apron. Images source: [1].

```
__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: 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: Naïve Sobel convolution

```
__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)];
        }
```

Figure: Sobel convolution using shared memory and separable filters

```
        __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: (cont'd.) 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: 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;
```

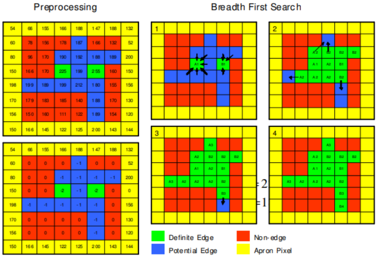
Figure: Edge thinning kernel

```
        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];
        }
}
```

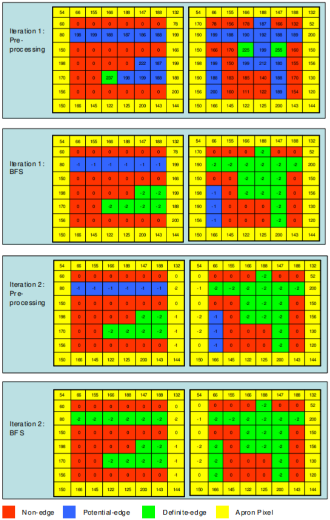
Figure: (cont'd.) Edge thinning kernel

```
#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: Double-thresholding kernel

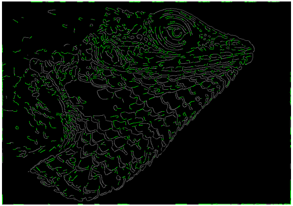


(a) Hysteresis edge tracking within a block (BFS in shared memory)

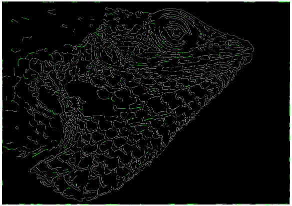


(b) Edge tracking across blocks (multiple passes)

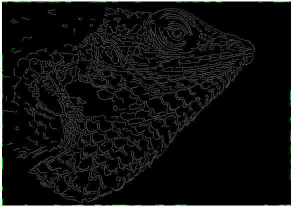
Figure: Hysteresis algorithm. Images source: [2].



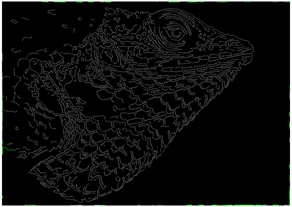
(a) 1 pass



(b) 2 passes



(c) 3 passes



(d) 4 passes

Figure: 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.

```
// 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;
```

Figure: Naïve hysteresis using global memory

```
                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: (cont'd.) 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;
```

Figure: Hysteresis using shared memory

```
                // 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]);
            }
        }
}
```

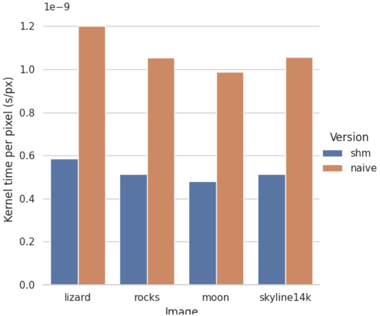
Figure: (cont'd) Hysteresis using shared memory

Test setup and objectives:

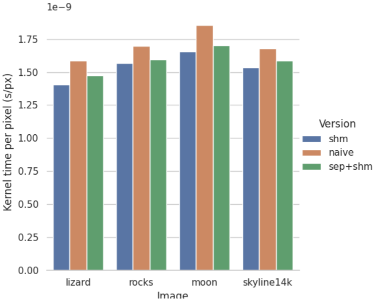
- NVIDIA GT740 (1GB) vs. Intel i5-7267U
- Compare all kernels with CPU equivalents
- Compare optimized/unoptimized kernels
- Compare accuracy

Kernel	Time Reduction (%)	Speedup (%)
blur	99.98 (opt); 99.3 (unopt)	500000; 14200
sobel	96	2400
edgethin	93	1300
threshold	86	600
hysteresis	29	40

Table: Speedups for CUDA kernels vs. CPU equivalent. Average image accuracy: $\approx 99\%$.

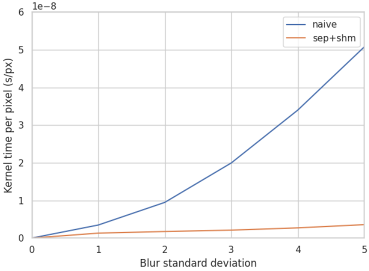


(a) Comparison of normalized kernel times of the two hysteresis implementations.

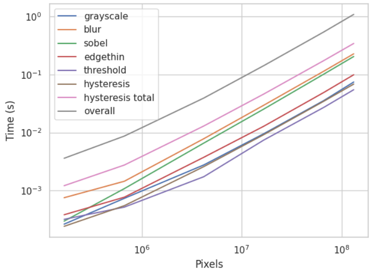


(b) Comparison of normalized kernel times of the three Sobel filter implementations.

Figure: Result of optimizing kernels. Plots are normalized by the number of pixels in the image.



(a) Comparison of the normalized kernel times of the two (Gaussian) convolution implementations for varying σ.



(b) Timings of the different kernels on images of different sizes.

Figure: Showing scalability of the two blur kernels for different blur sizes, and for all kernels over different image sizes.

References

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

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.