GAUSSIAN BLUR EFFECT

BASED ON WAVELET IMAGE PROCESSING

Yue Wang

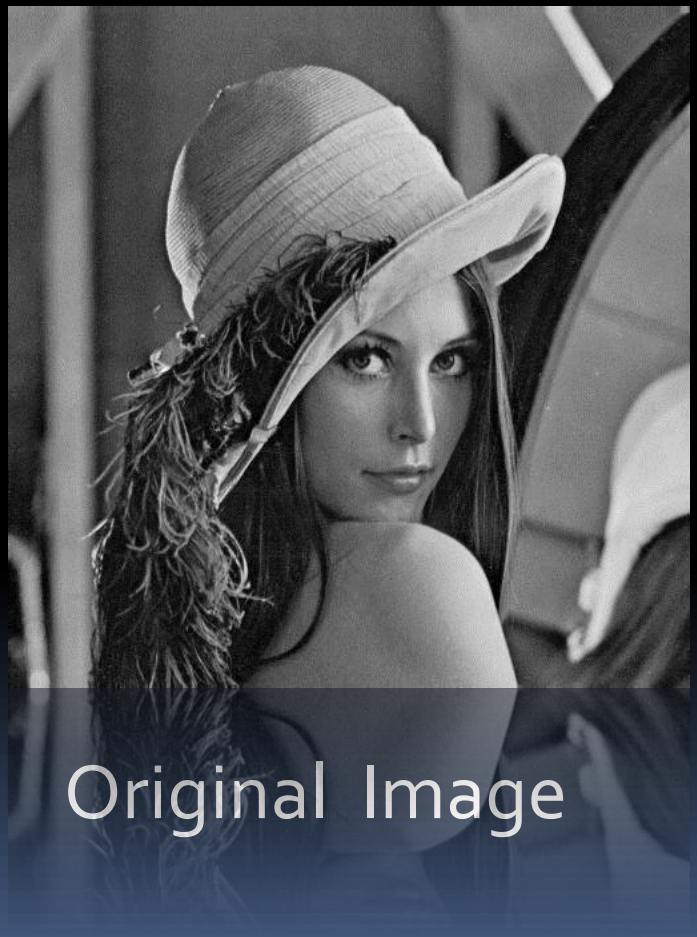
ID: 31085291

Introduction of Basic Knowledge

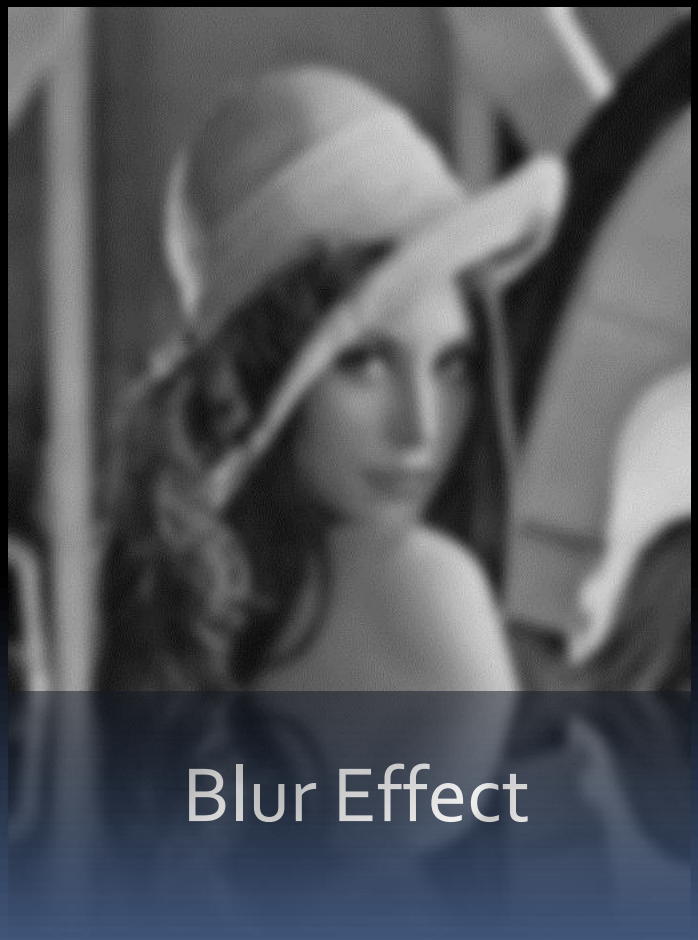
Gaussian Blur Effect

A Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the Bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination.

Example of Gaussian Blur



Original Image



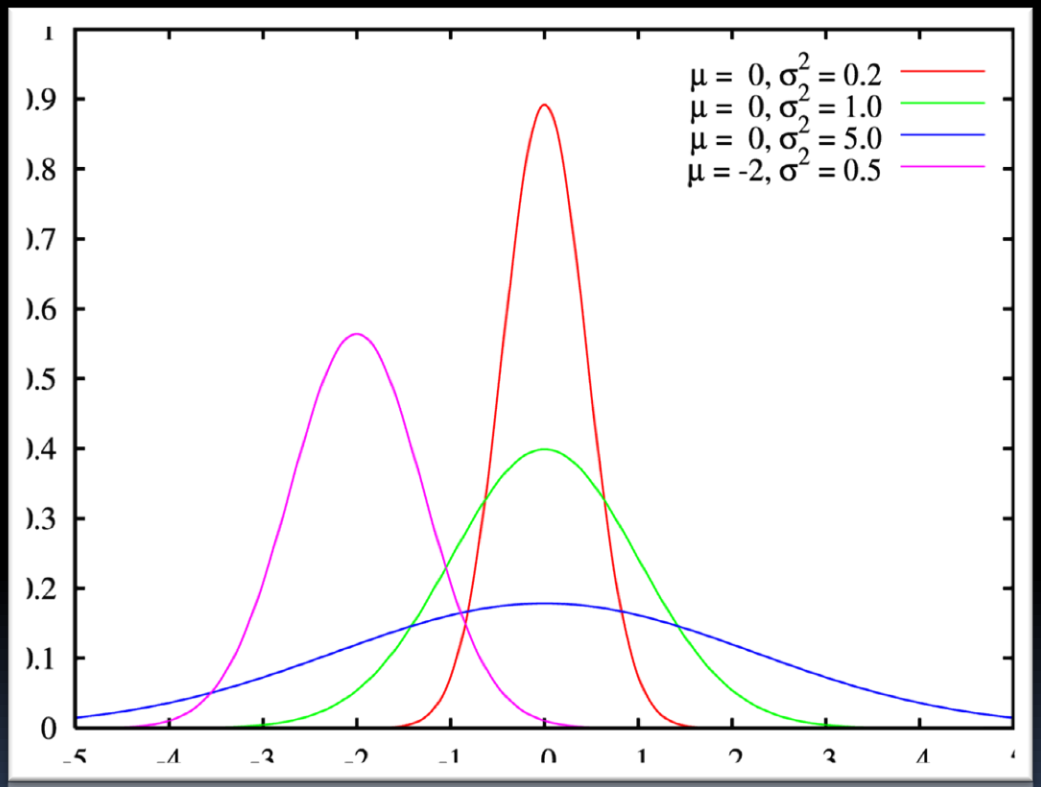
Blur Effect

Gaussian Function

In mathematics, a Gaussian function (named after Carl Friedrich Gauss) is a function of the form:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

Gaussian Curves with different σ



It is essentially a Low Pass Filter

Introduction to GPU & CUDA

- What is GPU

A graphics processing unit(GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display.

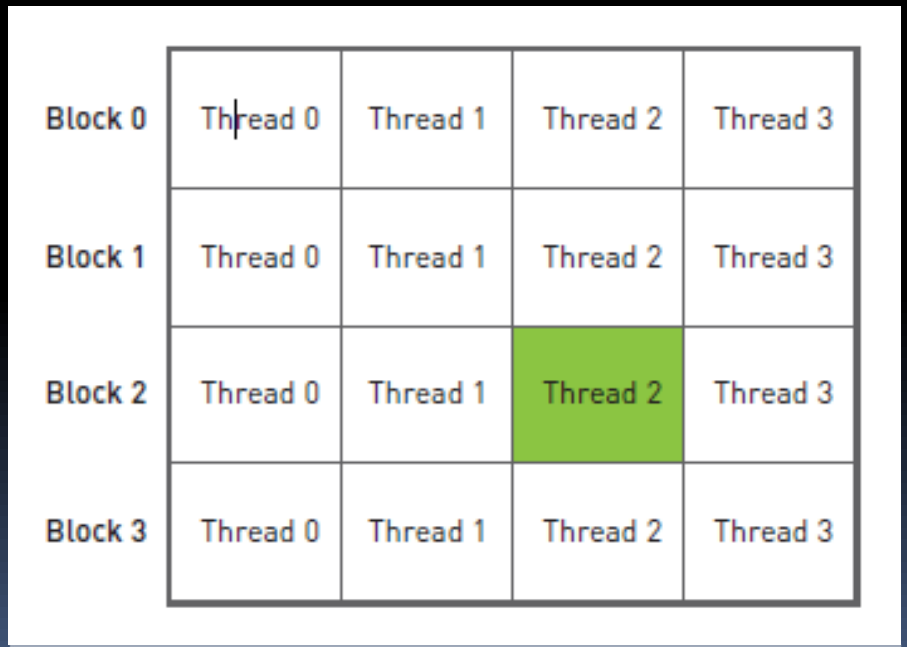
- What is CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia for graphics processing.

How CUDA works — 1

- Create Blocks

Declare i = blockIdx.x and j = blockIdx.y to create i * j blocks which will be operated parallely in GPU.



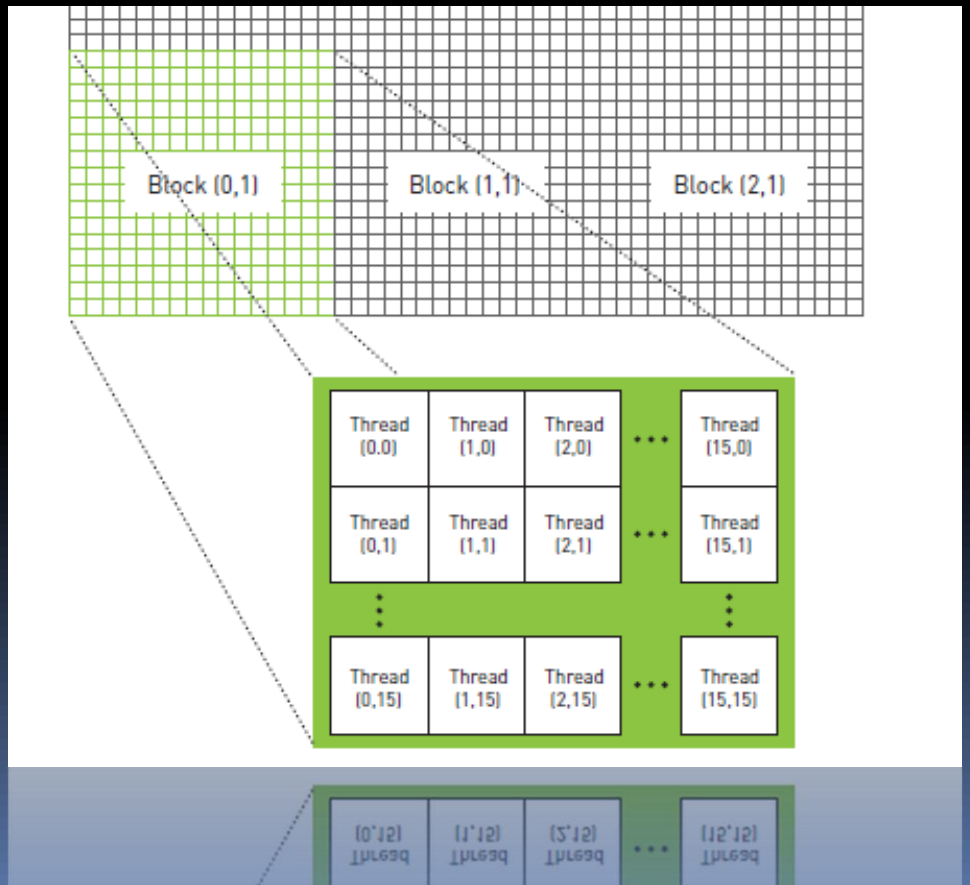
4 blocks on GPU

blocks id: 0-3

How CUDA works — 2

- Create Threads

Declare k = threadIdx.x, which based on the number of data that will be operated in each of the blocks



16*16 threads in each block

Implimentation with CUDA C

Major Steps

- Create Gaussian kernel vector;
- Multiply every n consecutive elements of every row in image with every n elements in vector in each blocks horizontally from left to right;
- Multiply every n consecutive elements of every column in image with every n elements in vector in each blocks vertically from up to down;
- Output the result.

Create Gaussian Kernel

- Based on the function: $f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$

  Coding is typed as following

```
        int sigma = 0;
        printf("\nint put the SIGMA of Gaussian Kernel: ");
        scanf("%d", &sigma);
        float scale = -0.5/(sigma*sigma);
        const float PI = 3.14159;
        float cons = 1/sqrt(-scale / PI);
        float sum = 0;
        int kcenter = M/2;
        printf("\nGaussian Kernel:\n");
        for (int i=0; i<M;i++)
        {
                int x = i - kcenter;
                a[i] = cons * exp(x * x * scale);
                sum += a[i];
//              printf("%f ", a[i]);
        }
        for (int i=0; i<M; i++)
        {
                a[i]/=sum;
                printf("%f ", a[i]);
        }
        }
```

Assign dimension of blocks

A sample [illegible] ls

(Just a 10 * 10 sample matrix created for experiment, since the console window does not supply for huge pixels display.)

```
// get the x and y indexes of the block
int i = blockIdx.x;
int j = blockIdx.y;
```

I, treated as index of row, assigned by the value of "blockIdx.x", whose range is from 0 to 9;

J, treated as index of column, assigned by the value of "blockIdx.y", whose range is also from 0 to 9;

Assign dimension of threads

The dimension of threads is assigned to 5, since the length of kernel is set initially by the code.

```
int k = threadIdx.x;
```

How to index

Every elements in 10*10 matrix are treated as center [illegible] cks. For instanc [illegible] t in the block[3

```
// calculate the indexes method
int x = i * N + j - size/2 +k;
int y = k;
int z = j * N + i;
```

b[3][4], treated as a result matrix, equals to a[3][4] and other 4 neighbor element multiplied by 5 element in kernel vector respectively:

b[3][4]=a[1][4]*c[0]+a[2][4]*c[1]+a[3][4]*c[2]+a[4][4]*c[3]+a[5][4]*c[5]

How to sum them up

```
if (k == 0){                                    //calculate all the blocks
        int sum_a = 0;
        int ctr = 0;
        float temp = 0;
        while (ctr<N){
                temp += cache[ctr];
                sum_a += a[ctr];
                ctr++;
        }
        c[z] = temp ;
        if (c[z] < 0)
                        c[z] = 0;
                else if (c[z] > 255)
                        c[z] = 255;
        c[z] = temp / sum_a;
}
}
```

Sum all 5 elements in every block

How to handle with boundary

For those blocks whose index are less than half of the size of kernel vector, there are not enough neighbor elements, they cannot be calculated correctly.

Take a[0][0] for example:

There is no a[-2][0] and a[-1][0], these two elements can be any arbitrary number in registers of GPU. If just add them directly, the result might be very crazy.

How to handle with boundary

If just ignore those boundary blocks, the result image is not quite acceptable. This will loss many pixels of image.





Check out the boundary carefully

How to handle with boundary

One effective way is add a discussion. Just add those elements that are multiplied with kernel vector.

```
for(i=0;i<N;i++){                                   //handle with bondary of left

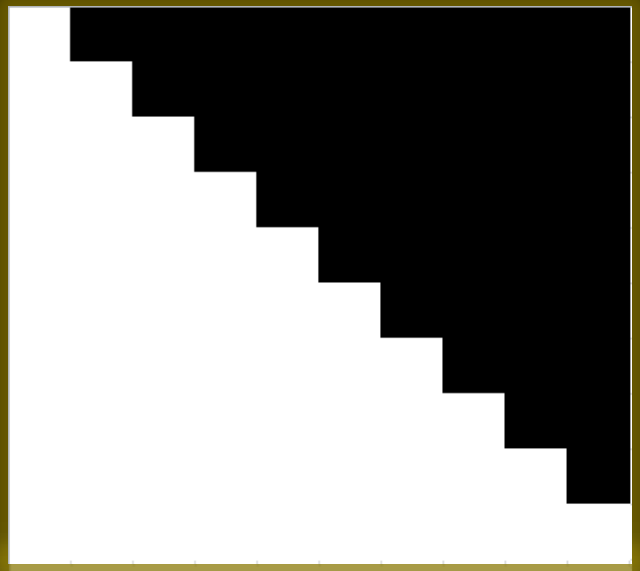
        for(j=0;j<size/2;j++){
                float sum_a=0;
                int temp=0;
                for(int ctr=size/2; ctr<size; ctr++){
                        temp += b[i*N +j-size/2+ctr]*a[ctr];
                        sum_a +=a[ctr];
                }
                c[j*N+i] = temp;
                c[j*N+i] = temp / sum_a;                    //transform matrix c
                if (c[j*N +i] < 0)
                        c[j*N +i] = 0;
                else if (c[j*N +i] > 255)
                        c[j*N +i] = 255;
        }

}
```

Advanced: change the coefficients in kernel vector

```
        int sigma = 0;
        printf("\nint put the SIGMA of Gaussian Kernel: ");
        scanf("%d", &sigma);
        float scale = -0.5/(sigma*sigma);
        const float PI = 3.14159;
        float cons = 1/sqrt(-scale / PI);
        float sum = 0;
        int kcenter = M/2;
        printf("\nGaussian Kernel:\n");
        for (int i=0; i<M;i++)
        {
                int x = i - kcenter;
                a[i] = cons * exp(x * x * scale);
                sum += a[i];
//              printf("%f ", a[i]);
        }
        for (int i=0; i<M; i++)
        {
                a[i]/=sum;
                printf("%f ", a[i]);
        }
        }
```

Sample1

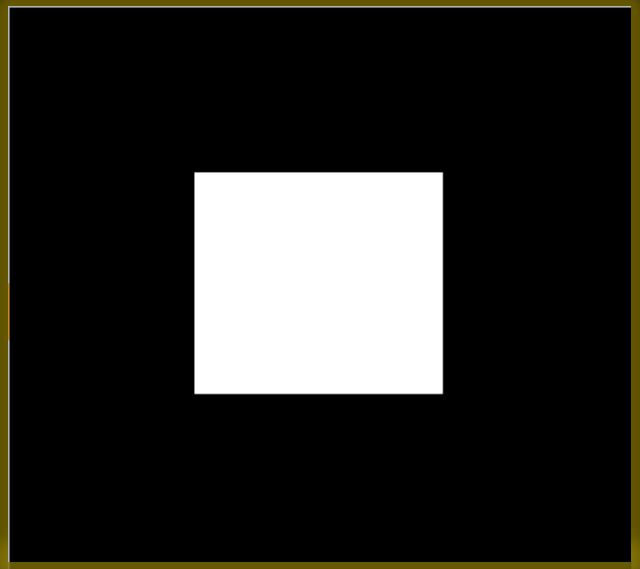


original image 1

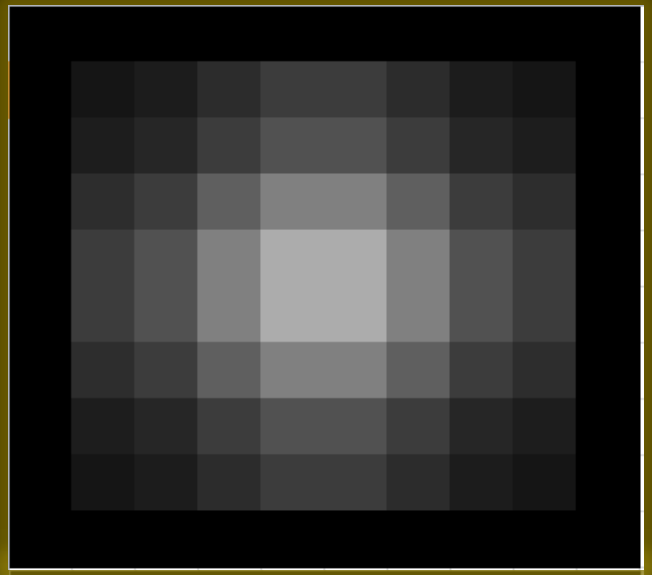


result image 1

Sample2



original image 2



result image 2

Reference

- Wikipedia, Gaussian function http://en.wikipedia.org/wiki/Gaussian_function
- CSDN, Summation of Gaussian Blur, zddmail http://blog.csdn.net/zddmail/article/details/7450033
- Gaussian Smoothing, Gaussian Blur, Gaussian Filter, hhygcy

  http://blog.csdn.net/hhygcy/article/details/4329056
- CUDA by Example An Introduction to General-Purpose GPU Programming, Jason Sanders & Edward Kandrot