High-Performance Programming With Graphic Cards
Laboratory Course

Canny edge detector

Lun-Yu Yuan - 3642753
Jatin Patel - 3644188

Universität Stuttgart – March 29, 2024

Project repository link: **GPU Project Repository**

1. Conceptual Design

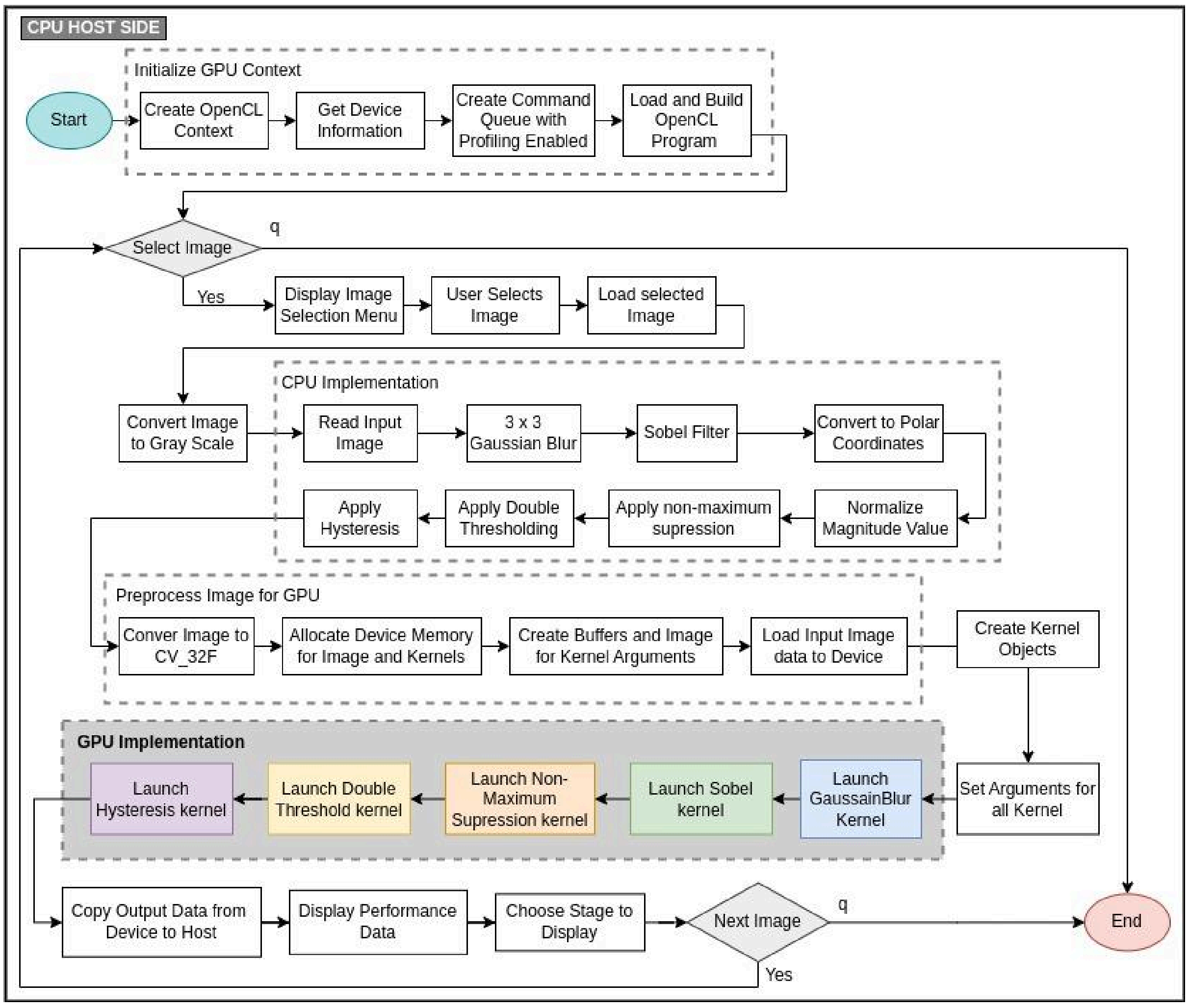
Canny Edge detection is implemented in five stages, which are shown as follows:

1) Smoothing the Imgae by applying a Gaussian Filter to eliminate noises.
2) Apply the Sobel filter to compute the horizontal and vertical gradient of the image.
3) Determine the local orientation of edges based on the gradient angles and then apply non-maximum suppression to suppress non-maximum gradient values by comparing them with neighbouring gradient magnitudes along the edge orientation.
4) Utilizing double thresholding to identify strong and weak edges based on pre-defined thresholds.
5) Tracking edges through hysteresis: First, use strong edges as the starting points and then follow weak edges connected to the strong edges to form continuous edge contours. After that, promote the weak edges to strong edges if they are connected to strong edges.

2. Implementation

In the Canny Edge Detection project, OpenCV is employed for image processing tasks such as loading, displaying images, and converting them to data types CV_32F and CV_8U, suitable for CPU and GPU processing, respectively. To facilitate the display of intermediary images during GPU processing, multiple buffers, utilizing the cv::Mat data type, are to hold the images generated at each processing stage. Upon completion of both CPU and GPU image processing, the duration of CPU and GPU processing, along with the time taken for data copying, are presented. Subsequently, a user interface is made available, allowing users to select and view their preferred processed image.

Note: *CPU implementation in our project is used from reference 3. We modularize the Canny Edge Detection algorithm in various functions for CPU Implementation.*


CPU HOST SIDE
Initialize GPU Context
Start
Create OpenCL Context
Get Device Information
Create Command Queue with Profiling Enabled
Load and Build OpenCL Program
Select Image
q
Yes
Display Image Selection Menu
User Selects Image
Load selected Image
CPU Implementation
Convert Image to Gray Scale
Read Input Image
3 x 3 Gaussian Blur
Sobel Filter
Convert to Polar Coordinates
Apply Hysteresis
Apply Double Thresholding
Apply non-maximum supression
Normalize Magnitude Value
Preprocess Image for GPU
Conver Image to CV_32F
Allocate Device Memory for Image and Kernels
Create Buffers and Image for Kernel Arguments
Load Input Image data to Device
Create Kernel Objects
GPU Implementation
Launch Hysteresis kernel
Launch Double Threshold kernel
Launch Non-Maximum Supression kernel
Launch Sobel kernel
Launch GaussainBlur Kernel
Set Arguments for all Kernel
Copy Output Data from Device to Host
Display Performance Data
Choose Stage to Display
Next Image
q
Yes
End

2.1. Buffer Implementation

To deal with the intermediary images and the final images, create multiple `cv::MAT` to store each image.

```
        Mat h_outputGpu ( countY, countX, CV_32F );
        Mat h_intermediate_blur ( countY, countX, CV_32F );
        Mat h_intermediate_sbl ( countY, countX, CV_32F );
        Mat h_intermediate_nms ( countY, countX, CV_32F );
        Mat h_intermediate_strong ( countY, countX, CV_32F );
        Mat h_intermediate_weak ( countY, countX, CV_32F );
```

After each kernel processes the image, the output image from the GPU will be copied back to the CPU buffer, for example:

```
// store Blur output image back to the host
        queue.enqueueReadImage ( bufferGBtoSBL, true, origin,
region, countX * sizeof ( float ), 0, h_intermediate_blur.data,
NULL );
```

2.2. Image data type conversion

2.2.1. CV_8U and CV_32F

In this project, there are two image data types are used, CV_32F (32-bit floating-point data type in OpenCV) and CV_8U (8-bit unsigned integer data type in OpenCV, 0-255.) Many GPUs are optimized for floating-point operations, which makes CV_32F a preferred choice for complex calculations. Thus, the conversion of CV_8U to CV_32F can prevent overflow and underflow errors from occurring in GPU calculation.

```
if ( img.type() != CV_32F ) {
            img.convertTo ( img, CV_32F, 1/255.0 );
            }
```

Following copying back the GPU image output into the host memory, converting the image from CV_32F to CV_8U makes the image compatible with standard display systems, allowing it to be viewed on screens. Also, CV_32F might result in pixel values outside the 0 - 255 range or in floating-point values. Converting to CV_8U involves scaling and possibly clipping the pixel values to fit within the 0-255 range, ensuring that the image data is properly normalized and can be displayed correctly. For example:

```
displayBlur.convertTo ( displayBlur, CV_8U );
```

2.2.2. cv::Mat and cl::Image2D

cv::Mat is a matrix data type used in OpenCV, a popular computer vision library. It is designed for efficient storage and manipulation of images and other forms of multidimensional data on the CPU.
cl::Image2D is part of the OpenCL framework, which is used for writing programs that execute across heterogeneous platforms consisting of CPUs, and GPUs.

Thus, the conversion of cv::Mat to cl::Image2D is necessary for the GPU processing. For example:

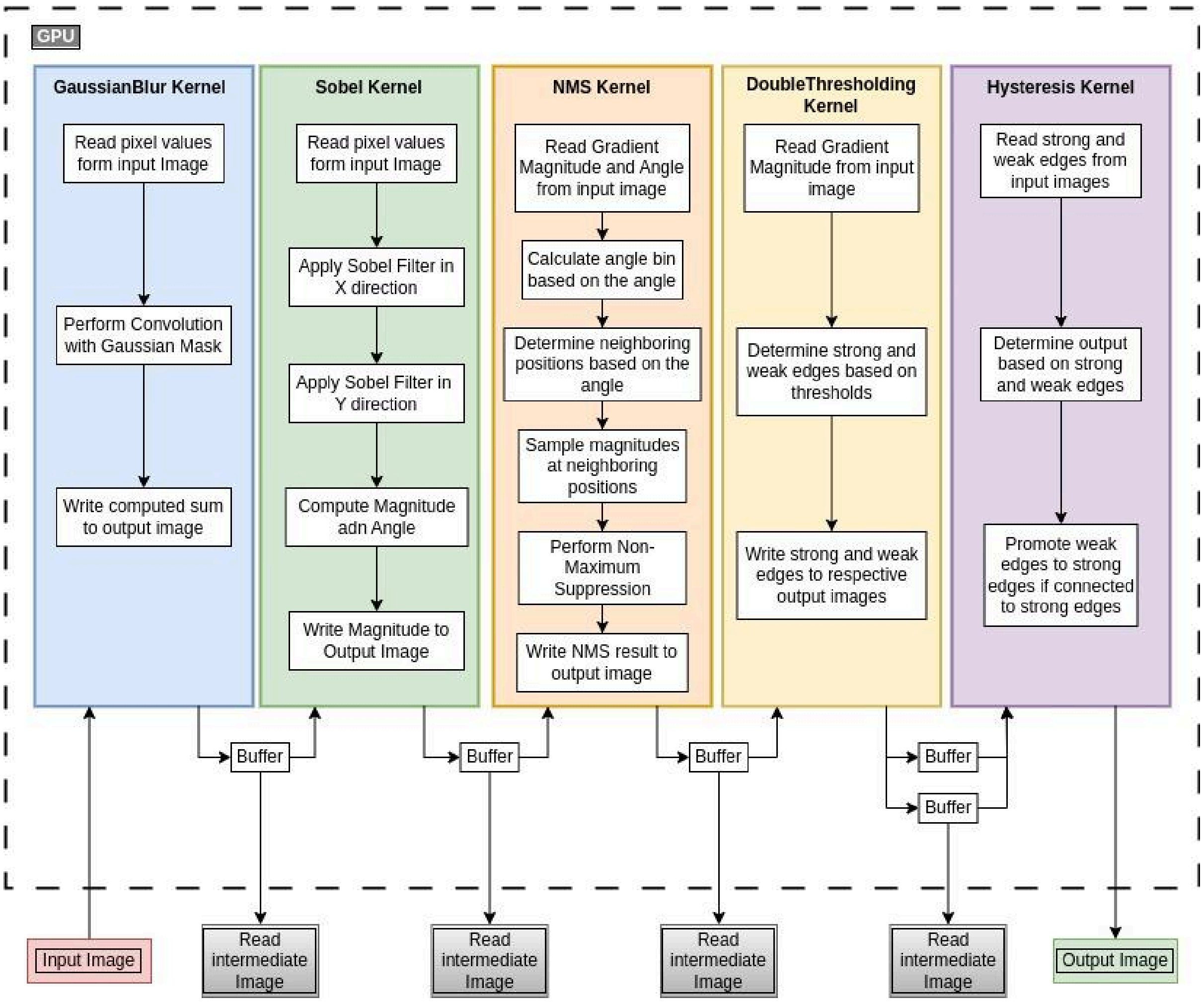
```
cl::Image2D image;
image = cl::Image2D ( context, CL_MEM_READ_WRITE,cl::ImageFormat (
CL_R, CL_FLOAT ), countX, countY );

// Transfer input image data from host to device by writing it to
```

```
an OpenCL Image2D object
queue.enqueueWriteImage ( image, true, origin, region,
countX * sizeof ( float ), 0, imgVector.data(), NULL, &copy1 );
```

2.3. canny.cl

In kernel implementation, our implementation is structured into several key kernels, each performing specific tasks in the edge detection workflow: Gaussian Blur, Sobel Filtering, Non-Maximum Suppression, Double Thresholding, and Hysteresis. Below is a detailed description of each stage.



GPU
GaussianBlur Kernel
Read pixel values form input Image
Perform Convolution with Gaussian Mask
Write computed sum to output image
Sobel Kernel
Read pixel values form input Image
Apply Sobel Filter in X direction
Apply Sobel Filter in Y direction
Compute Magnitude adn Angle
Write Magnitude to Output Image
NMS Kernel
Read Gradient Magnitude and Angle from input image
Calculate angle bin based on the angle
Determine neighboring positions based on the angle
Sample magnitudes at neighboring positions
Perform Non-Maximum Suppression
Write NMS result to output image
DoubleThresholding Kernel
Read Gradient Magnitude from input image
Determine strong and weak edges based on thresholds
Write strong and weak edges to respective output images
Hysteresis Kernel
Read strong and weak edges from input images
Determine output based on strong and weak edges
Promote weak edges to strong edges if connected to strong edges
Buffer
Buffer
Buffer
Buffer
Buffer
Input Image
Read intermediate Image
Read intermediate Image
Read intermediate Image
Read intermediate Image
Output Image

2.3.1. Gaussian Blur

To smooth the input image and reduce noise that can affect edge detection, use a Gaussian mask to perform convolution on it. Each pixel's new value is calculated by applying the Gaussian mask, thereby smoothing the image and preparing it for edge detection.

This OpenCL kernel, GaussianBlur, applies a Gaussian blur to a 2D input image. It is defined to accept four parameters: a `__read_only image2d_t` input image for the original, unblurred picture, a `__constant float*` mask which represents the Gaussian kernel weights, a `__write_only image2d_t` output image to hold the blurred result, and a `__private int` indicating the radius of the Gaussian mask, which determines the size of the area around each pixel to consider for blurring.

For each pixel, it calculates a weighted sum of the pixel values in its neighbourhood, defined by the mask size. This involves a nested loop over the mask dimensions, where for each relative position (a, b) within the mask, it reads the corresponding pixel value from the input image, multiplies it by the Gaussian weight from the mask, and adds this to the sum. The resulting sum, which is the blurred value of the current pixel, is then written to the corresponding position in the output image. The operation effectively blurs the image by averaging each pixel's value with its neighbours' values, weighted by the Gaussian mask.

```
sum += mask[(a + maskSize) * (maskSize * 2 + 1) + (b + maskSize)]
* imageValue;
```

2.3.2. Sobel Filtering

Gradient Calculation:
Horizontal Gradient (Gx): The kernel calculates the horizontal gradient at the current pixel by applying the Sobel filter in the X direction. This involves summing the weighted values of the pixel and its immediate neighbours in the horizontal direction. The weights are -1, -2, and 1, designed to highlight horizontal edges.

Vertical Gradient (Gy): Similarly, it calculates the vertical gradient by applying the Sobel filter in the Y direction. This step uses the same neighbours but weights them to emphasize vertical edges.

```
float Gx = -1 * getValueImage(blurredImage, x - 1, y - 1, sampler)
               -2 * getValueImage(blurredImage, x - 1, y, sampler)
               -1 * getValueImage(blurredImage, x - 1, y + 1,
sampler)
               +1 * getValueImage(blurredImage, x + 1, y - 1,
sampler)
               +2 * getValueImage(blurredImage, x + 1, y, sampler)
               +1 * getValueImage(blurredImage, x + 1, y + 1,
sampler);

    // Sobel filter in Y direction (vertical edges)
    float Gy = -1 * getValueImage(blurredImage, x - 1, y - 1,
sampler)
               -2 * getValueImage(blurredImage, x, y - 1, sampler)
               -1 * getValueImage(blurredImage, x + 1, y - 1,
sampler)
                    +1 * getValueImage(blurredImage, x - 1, y + 1,
sampler)
                    +2 * getValueImage(blurredImage, x, y + 1, sampler)
                    +1 * getValueImage(blurredImage, x + 1, y + 1,
sampler);
```

Magnitude and Angle:
The kernel calculates the magnitude of the gradient vector (magnitude) as the Euclidean norm (hypot(Gx, Gy)) of the horizontal and vertical gradients. This magnitude indicates the strength of the edge at the pixel.

2.3.3. Non-Maximum Suppression

Gradient Retrieval: It reads the gradient values at the current position from bufferSBLtoNMS. The gradient values are assumed to be stored in the first two channels (x and y components of a float4), representing the gradient magnitude and possibly its direction.

Angle Adjustment and Binning: The angle is adjusted to a range of 0 to 360 degrees (if necessary) and then mapped to one of eight possible bins representing the primary and secondary compass directions (N, NE, E, SE, S, SW, W, NW). This binning simplifies the comparison of the current pixel's gradient magnitude with its neighbours along the gradient direction.

Neighbouring Positions Determination: Depending on the angle bin, two neighbouring positions (pos1 and pos2) relative to the current pixel are determined. These positions are chosen such that they lie approximately along the gradient direction, enabling comparison of gradient magnitudes in a line perpendicular to the edge direction.

```
int2 pos1, pos2;
    switch (angleBin) {
        case 0:  pos1 = (int2)(-1, 0); pos2 = (int2)(1, 0); break;
        case 1:  pos1 = (int2)(-1, -1); pos2 = (int2)(1, 1);
break;
        case 2:  pos1 = (int2)(0, -1); pos2 = (int2)(0, 1); break;
        case 3:  pos1 = (int2)(-1, 1); pos2 = (int2)(1, -1);
break;
        case 4:  pos1 = (int2)(-1, 0); pos2 = (int2)(1, 0); break;
        case 5:  pos1 = (int2)(-1, -1); pos2 = (int2)(1, 1);
break;
```

```
        case 6:  pos1 = (int2)(0, -1); pos2 = (int2)(0, 1); break;
        case 7:  pos1 = (int2)(-1, 1); pos2 = (int2)(1, -1);
break;
    }
```

Magnitude Comparison (NMS): The magnitudes of the gradients at the two neighbouring positions are sampled. The current pixel's magnitude is retained (i.e., written to bufferNMStoDT) only if it is greater than or equal to the magnitudes at both neighbouring positions, ensuring that only the strongest edges are highlighted. Otherwise, the magnitude is suppressed (set to 0).

2.3.4. Double Thresholding

Edge Classification:
The kernel first initializes variables strongVal and weakVal to 0.0f, representing the absence of an edge.
It then compares the gradient magnitude against the thresholds magMax and magMin.
If the magnitude exceeds magMax, the pixel is classified as a strong edge, and strongVal is set to 1.0f.
If the magnitude is between magMin and magMax, the pixel is classified as a weak edge, and weakVal is set to 1.0f.

```
    if (gradientMagnitude > magMax) {
        strongVal = 1.0f; // Strong edge
    } else if (gradientMagnitude > magMin) {
        weakVal = 1.0f; // Weak edge (candidate for hysteresis)
    }
```

2.3.5. Hysteresis

Initial Output Setting: The output for the current pixel starts off as the value read from a strong image. This means if a pixel is a strong edge, it will automatically be included in the final edge map.

Weak Edge Examination: If the current pixel is not a strong edge (strong == 0.0f) but is a weak edge (weak != 0.0f), the kernel examines its 8 neighbouring pixels to see if any are strong edges. This examination skips the centre pixel (the current pixel) to only consider the neighbours.

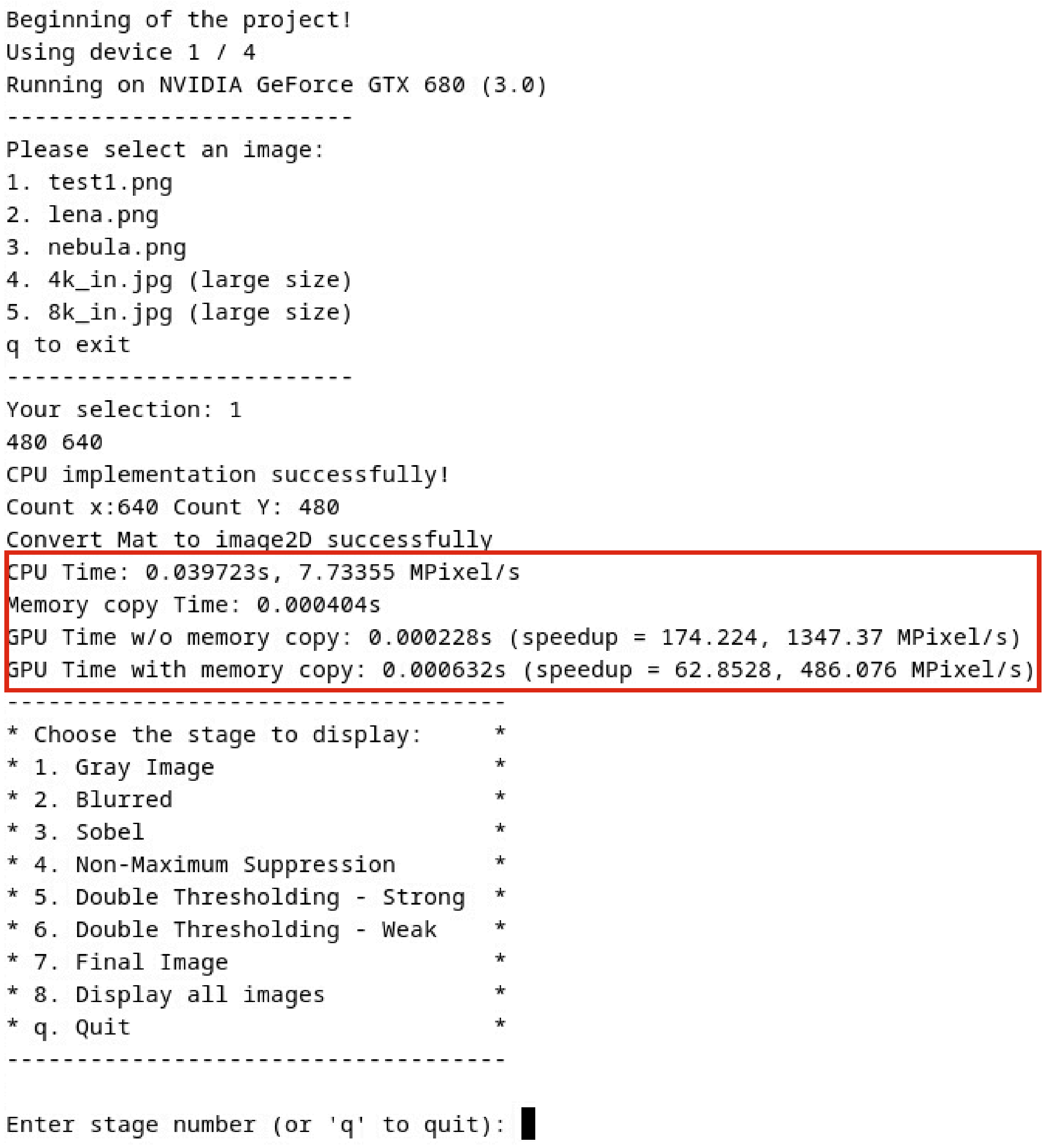
Promotion to Strong Edge: If any neighbouring pixel is found to be a strong edge (neighborStrong != 0.0f), the current weak edge pixel is "promoted" to a strong edge by setting output to 1.0f. This promotion process reflects the principle that weak edges directly connected to strong edges are likely part of true edges and should be preserved.

```
float output = strong; // Start with strong edges

    if (strong == 0.0f && weak != 0.0f) {
        // Examine the 8 neighbors
        for (int dx = -1; dx <= 1; dx++) {
            for (int dy = -1; dy <= 1; dy++) {
                if (dx == 0 && dy == 0) continue; // Skip the
center pixel
                float neighborStrong = read_imagef(strongImg,
sampler, (int2)(x + dx, y + dy)).x;
                if (neighborStrong != 0.0f) {
                    output = 1.0f; // Promote to strong edge
                    break;
                }
            }
        }
    }
```

3. Result

3.1. CPU and GPU Processing Time Comparison



```
Beginning of the project!
Using device 1 / 4
Running on NVIDIA GeForce GTX 680 (3.0)
-------------------------
Please select an image:
1. test1.png
2. lena.png
3. nebula.png
4. 4k_in.jpg (large size)
5. 8k_in.jpg (large size)
q to exit
-------------------------
Your selection: 1
480 640
CPU implementation successfully!
Count x:640 Count Y: 480
Convert Mat to image2D successfully
CPU Time: 0.039723s, 7.73355 MPixel/s
Memory copy Time: 0.000404s
GPU Time w/o memory copy: 0.000228s (speedup = 174.224, 1347.37 MPixel/s)
GPU Time with memory copy: 0.000632s (speedup = 62.8528, 486.076 MPixel/s)
------------------------------------
* Choose the stage to display:     *
* 1. Gray Image                    *
* 2. Blurred                       *
* 3. Sobel                         *
* 4. Non-Maximum Suppression       *
* 5. Double Thresholding - Strong  *
* 6. Double Thresholding - Weak    *
* 7. Final Image                   *
* 8. Display all images            *
* q. Quit                          *
------------------------------------

Enter stage number (or 'q' to quit): █
```

The highlighted section within the red rectangle indicates the performance metrics for image processing tasks, showcasing the efficiency of CPU versus GPU processing.

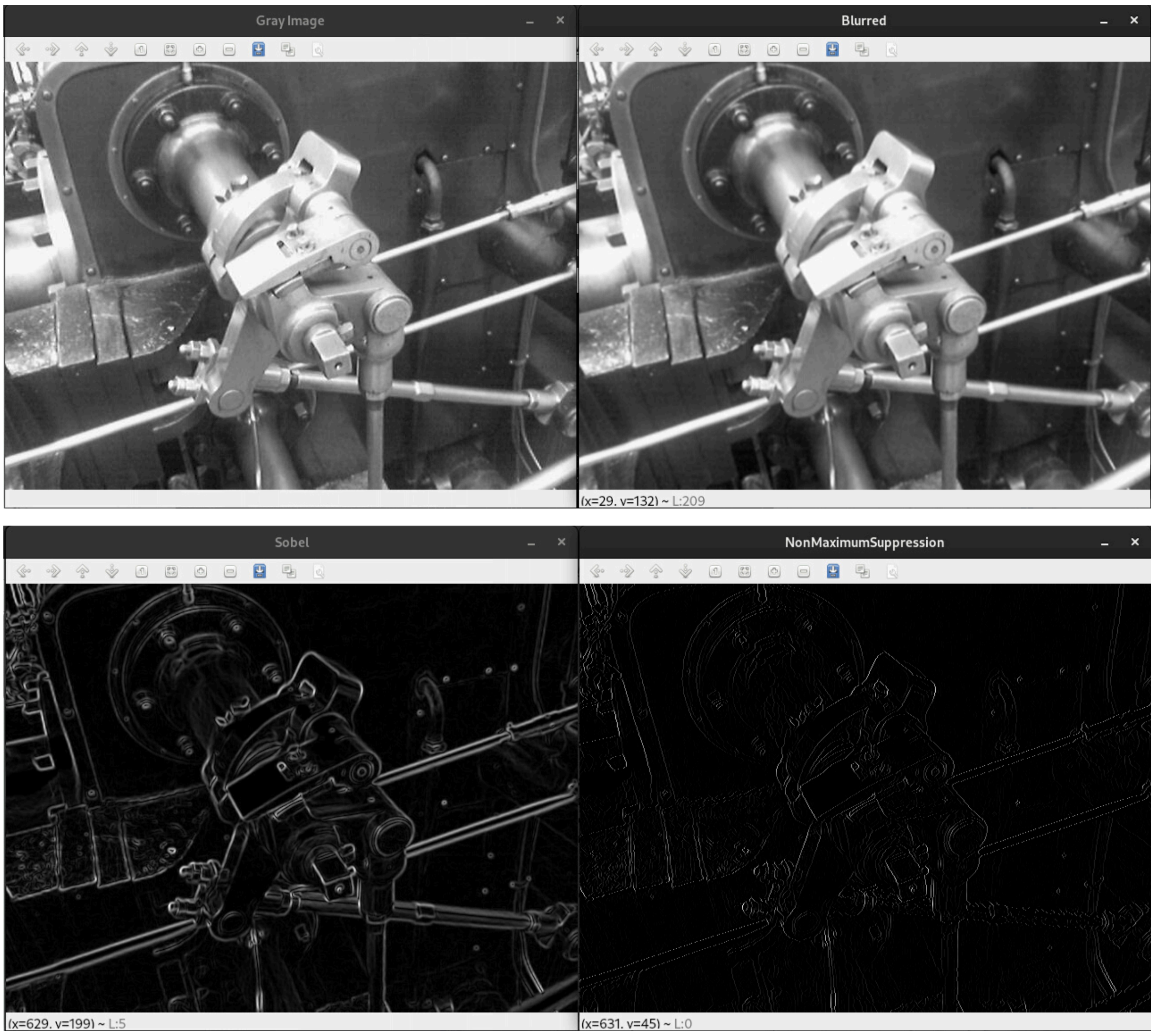
CPU Model: **Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz**
GPU Model: **NVIDIA Corporation GK104 [GeForce GTX 680]**

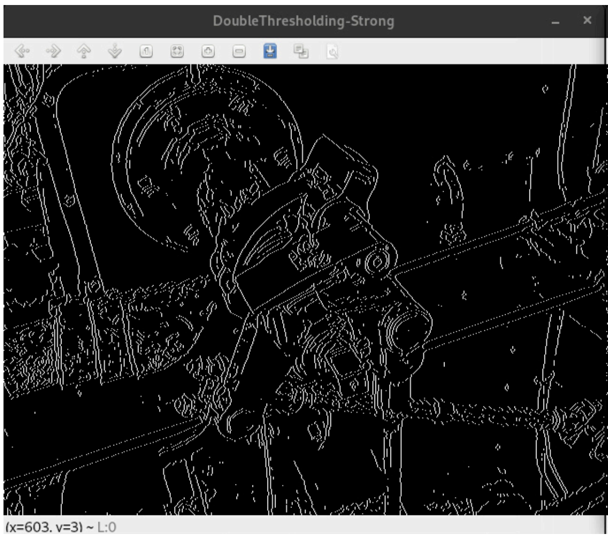
```
CPU Time: 0.039723s, 7.73355 MPixel/s
```

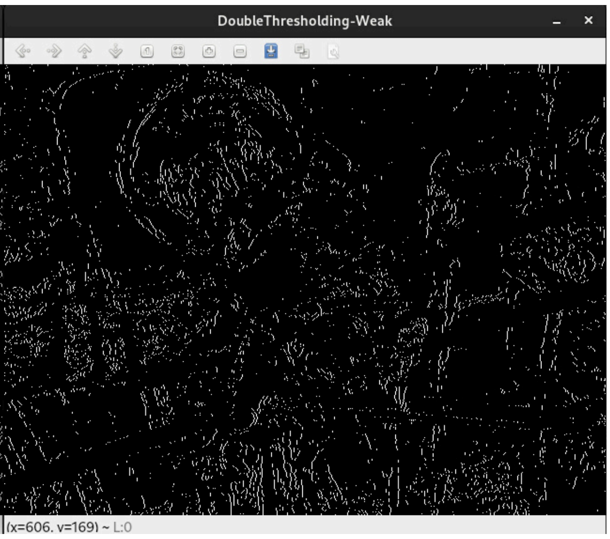
```
Memory copy Time: 0.000404s
GPU Time w/o memory copy: 0.000228s (speedup = 174.224, 1347.37
MPixel/s)
GPU Time with memory copy: 0.000632s (speedup = 62.8528, 486.076
MPixel/s)
```

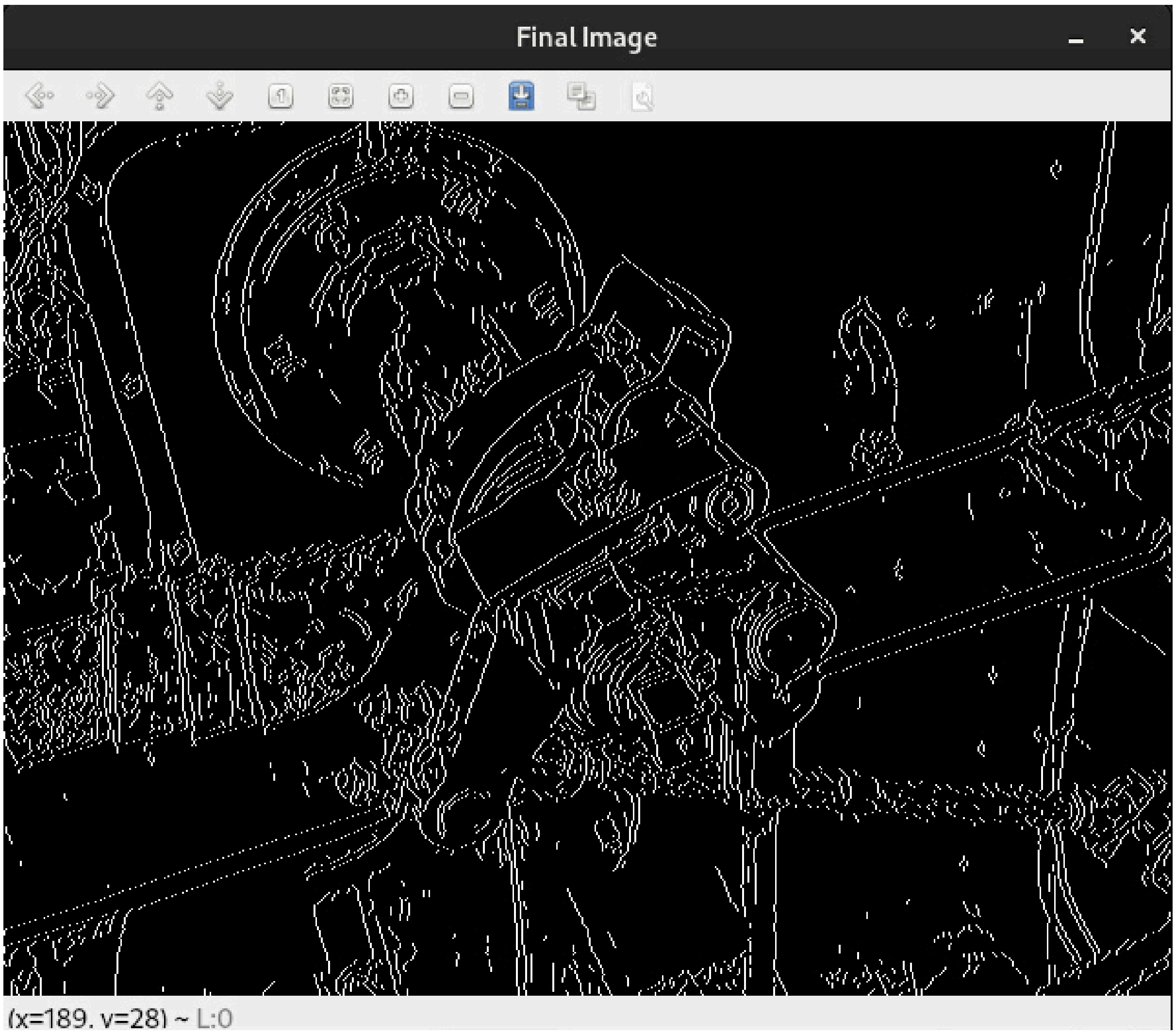
the GPU demonstrates a remarkable efficiency, being 174.224 times quicker than the CPU when memory copy time is excluded, and 62.8528 times faster when memory copy time is included.

3.2. Output Images




DoubleThresholding-Strong
(x=603, y=3) ~ L:0


DoubleThresholding-Weak
(x=606, y=169) ~ L:0


Final Image
(x=189, y=28) ~ L:0

4. References

1. https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html
2. https://en.wikipedia.org/wiki/Canny_edge_detector
3. https://github.com/akwCode/CED/blob/main/CED.cpp
4. https://github.com/smistad/OpenCL-Gaussian-Blur/tree/master
5. https://bashbaug.github.io/OpenCL-Docs/pdf/OpenCL_API.pdf