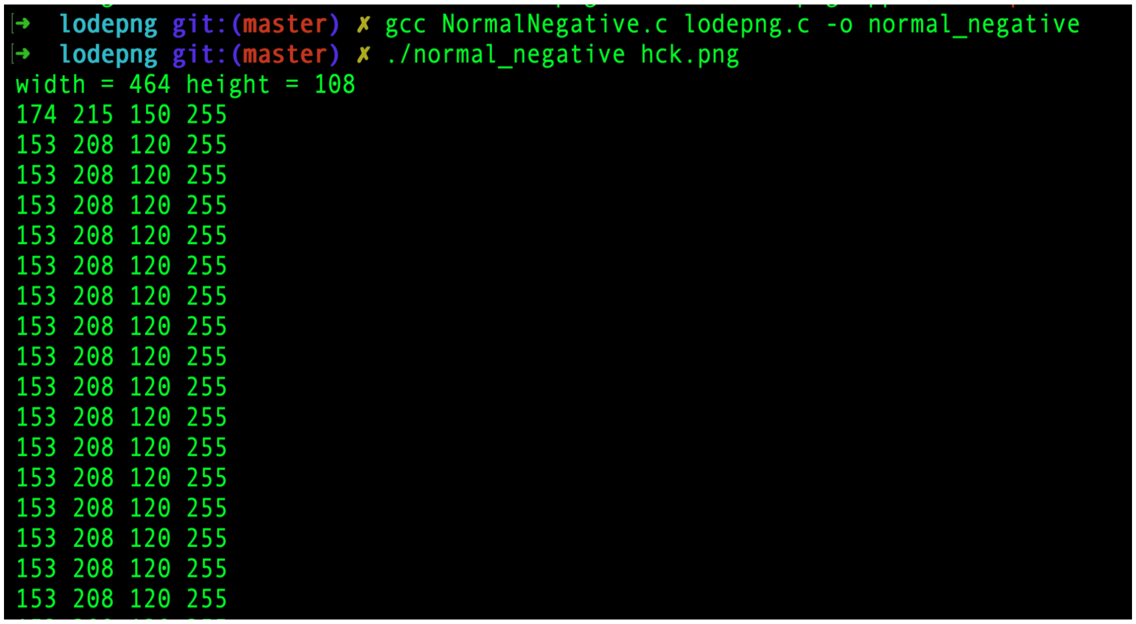
Pratik Dhimal
2407779

We are making negative image using these three techniques:

1. CPU(No multithreading)

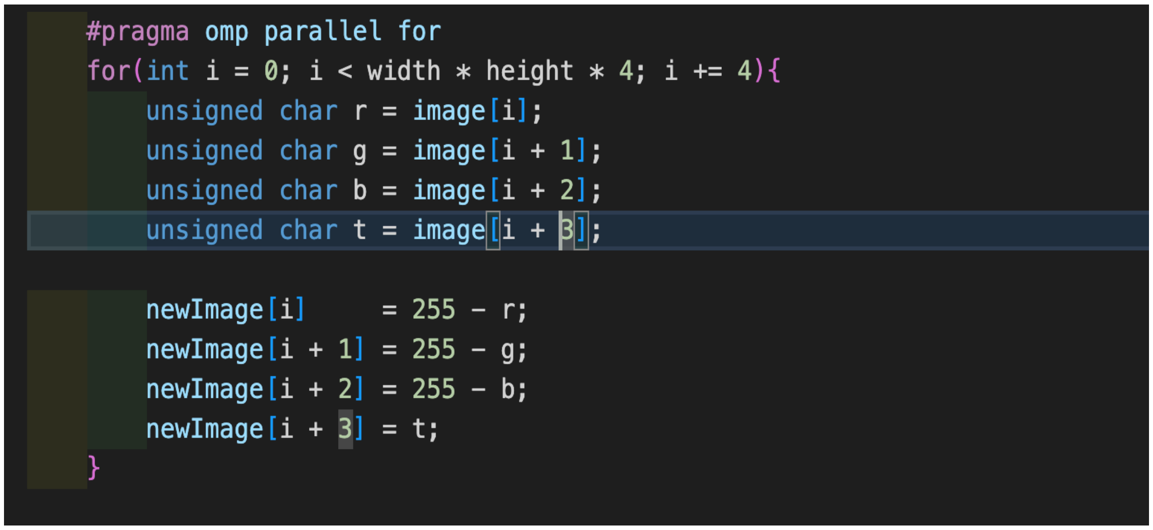
   ⇒we'll write a **single threaded program** that takes in the image and decodes it using lodepng_decode32_file() . This decoded image is nothing but pixels value in a 1-D array, all the pixel values are dumped into that array. Now we do our necessary operation i.e. to change the pixels to make it negative. For negative we subtract the pixel value of RGB not T, from 255 to get negative pixels. Lastly, we use the lodepng_encode32_file() function to encode the array values to a png image. We have displayed the width(pixels) and height(pixels) of the image and also each pixel value for visualization purposes. Finally, we have altered the pixel value and again formed an image which is negative.



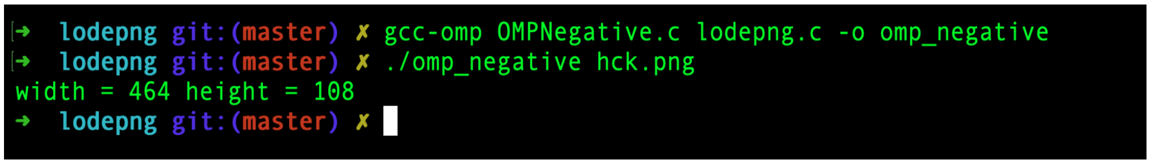
```
→  lodepng git:(master) ✗ gcc NormalNegative.c lodepng.c -o normal_negative
→  lodepng git:(master) ✗ ./normal_negative hck.png
width = 464 height = 108
174 215 150 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
153 208 120 255
```

2. CPU (Multithreading using openMP)

⇒ In this programme the whole logic for decoding and encoding is completely same as 1 but to manipulate the pixel we have used **#pragma omp parallel for** so that we utilize all cores of our cpu and **try to achieve parallelism.** When we use **#pragma omp parallel for** the compiler automatically divides each iteration of loop to threads, for example if we have 8 core cpu then it makes 8 threads and divides the load. Thus by this way we can use multithreading in CPU to solve our problem of making a negative image.



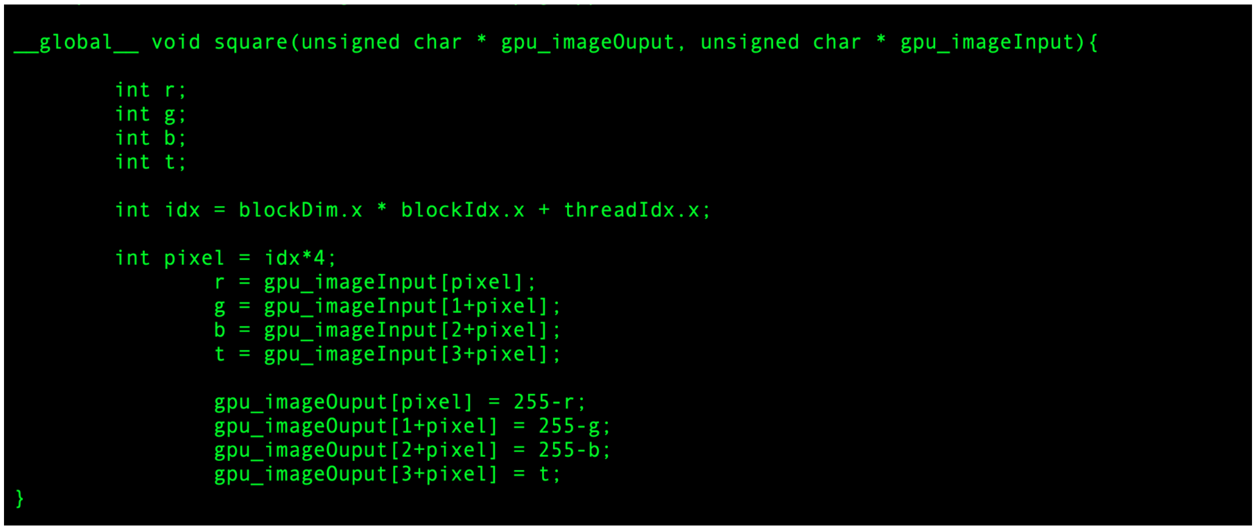
```
#pragma omp parallel for
for(int i = 0; i < width * height * 4; i += 4){
    unsigned char r = image[i];
    unsigned char g = image[i + 1];
    unsigned char b = image[i + 2];
    unsigned char t = image[i + 3];

    newImage[i]     = 255 - r;
    newImage[i + 1] = 255 - g;
    newImage[i + 2] = 255 - b;
    newImage[i + 3] = t;
}
```



```
➜  lodepng git:(master) ✗ gcc-omp OMPNegative.c lodepng.c -o omp_negative
➜  lodepng git:(master) ✗ ./omp_negative hck.png
width = 464 height = 108
➜  lodepng git:(master) ✗
```

3. GPU (Cuda)

⇒ This program utilizes the power of GPU cuda cores to perform the pixel manipulation logic very fast so that for a larger image with more pixels the time of pixel manipulation reduces drastically. Here in this code the decoding and encoding process is completely same just we have utilized the kernel function with config <<<width,height>> which created block equals to the width(pixels) and each block will have threads equal to the height so in thai way we are giving each thread a set of pixels(RGBT) to manipulate. The logic remains same subtracting from 255 to get the negative image.



```
__global__ void square(unsigned char * gpu_imageOuput, unsigned char * gpu_imageInput){

        int r;
        int g;
        int b;
        int t;

        int idx = blockDim.x * blockIdx.x + threadIdx.x;

        int pixel = idx*4;
                r = gpu_imageInput[pixel];
                g = gpu_imageInput[1+pixel];
                b = gpu_imageInput[2+pixel];
                t = gpu_imageInput[3+pixel];

                gpu_imageOuput[pixel] = 255-r;
                gpu_imageOuput[1+pixel] = 255-g;
                gpu_imageOuput[2+pixel] = 255-b;
                gpu_imageOuput[3+pixel] = t;
}
```