UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

6CS005 High Performance Computing

Portfolio Part 2

Student Name: Pratik Dhimal

University ID: 2407779

Group: L6CG1

Tutor: Yamu Poudel

Submission Date: 11th January, 2026

Table of Contents

1. Task3: Password Cracking

This task is based on the implementation of the password cracking application built on the GPU platfor. Its implementation is done according to the particular conditions of memory management and kernel running, as well as data transfer. The overall flow of the programm is as follows.

1.1. Generate encrypted password in the CPU

In this process the **PasswordGenerator.c** file generates password in **format aa11** where first two character is always lower cased alphabet and last 2 is always numeric valus. Then the 4 letter password is encrypted to a 10 letter password with simple asci conversion techniques. This password is than stored in a textfile as shown in **Figure 1.**

1.2. Hash the encrypted password using SHA512 Algorithm

As showin in **Figure 2** , In this process the build in function from **crypt()** is used to hash the password which wash encrypted in above step. This function comes from **crypt.h header file.** What happens here is the program goes through each line of the initially encrypted password and than hashes it and store to a output text file as specified in the comman line.

1.3. Hash all the possible password into SHA512.

As shown in **Figure 3**, for the first time we utilized the power GPU to calcualte all the possible combination of our passwords and than encrypt it usign the same way like we did in the first step. So now we'll have all the possible combination of **encrypted but not hashed passwords** which is **67600** in our case. Than as shown in **Figure 2,** the same **CPU function** is used to hash the 67600 passwords. Thus finally we'll have 67600 properly hashed passwords usign the same algorithm as the target file.

1.4. Crack the password and store to file

As shown in **Figure 4, brute-force** search is initially performed on the GPU with all possible passwords generated correspondingly to the passwords by all possible password combinations and the encrypted/ hash values are stored in a large look-up table of 67,600 encrypted hash values. All the items in the table are distinct passwords based on a predetermined pattern and encrypted password. During the cracking, the CUDA kernel is executed to ensure that each thread in the GPUs has a single target hash. The thread then performs a comparison of its target hash against all the precalculated hashes in the table in sequence, the brute-force step of comparison. In case of a match, the matching table index is transformed back to the original plaintext password and the result is stored in the output; when no match is found after searching the entire table, a dummy(XXX) is stored instead. This scheme decouples the generation step of brute-force with the comparison step, thus enabling the costly password space to be calculable only once and then reused successfully to crack numerous target hashes at once.

1.5. Memory deallocation – freeing memory appropriately

Lastly as shown in **Figure 5** , all the GPU and CPU memory is freed to make sure memory leak doesn't happen.

2. Task 4: Sobel Edege Detection

2.1. Read image into CPU memory and store appropriately

In this step I have read the image file into CPU memory using then **lodepng_decode32_file()** function. This function decodes the image to its raw pixels form (RGBA for each pixels) and also gives the **width and heigth in pixels**. Later in the process, we'll need to convert back the pixels value to image so we've created a memory space named **h_out_rgba** using the maclloc() function which will be used later**.** All of this process is given in **Figure 6**

2.2. Allocate GPU memory according to image size and transfer data to GPU

2.2.1. Allocating GPU Memory

Here **cudMalloc()** is used to allocate memory in the GPU, we have allocated memory for input rbga, gray converted, edeges detected, and output rgba for loding raw decoded rgba, storing converted gray values (one per pixel), detected edeges (one per pixel), output rgba to store the final manipulated rgba value respectively as shown in **Figure 7** In my case the **hck.png image is of 464 x 108 pixels** and as a singel **unsigned character holds 1 byte** of memory so the memory allocation looks soething like this as showin in table below:

Input RGBA	200448 bytes
GrayScale	50112 bytes
Edeges	50112 bytes
OutPut RGBA2	200448 bytes

2.2.2. Data Transfer to GPU

Now we have allocated sufficient memory in the GPU, we need to transfer data to it. We just transfer the **raw decoded pixel value** to the GPU as all other memory address will be filled by GPU itself by performing pixel manupulation. As shown in **Figure 9,** cudeMemcpy() function is used to transfer the data to the GPU we need to keep in mind the last argument is the direaction and it must be **cudaMemcpyHostToDevice** as we are copying from host(CPU) to device (GPU). How we have the required data in the device.

2.3. Edges detected correctly via the CUDA kernel function

2.3.1. RBGA to GrayScale Kernel

In this kernel we pass all the raw rgba value and it converts to **grayscale intensity** value using the human eye adaptive formula for grayscale based on luminiosity weights rather than a normal average method we used to use in the past as showin in **Figure 8.**

2.3.2. Edege Detection Kernel

As showin in **Figure 10 ,**this kernel performs the main logic of edege detection. Here we have two gradient **Gx and Gy** which is used to mease the rate of change of intensities in horizontal and vertical direction respectively.I t wokrs by multipling each value of the **gradient kernel** with its corrosponding grayscale value, and this step is repeated for each and every value of the grayscale intensity. So now we have both the gradiet in horizontal and vertical direction, then, the formula $\sqrt{Gx^2 + Gy^2}$ is used for the effective magnitude of intensity change. Lastly, the edeges values are populated to the **d_edges** variable.

2.3.3. Detected edeges to final RGBA kernel

We had one value per pexel in the gray scale intensity value now this kernel populates the value to all the RGB to make a perfect set of RGBA value as shown in **Figure 8 ,**the A (transparency) is set to 255 for a perfectly opaque imag. This makes sure we can later use the perfect set of RGBA to generatre output edege detected image.

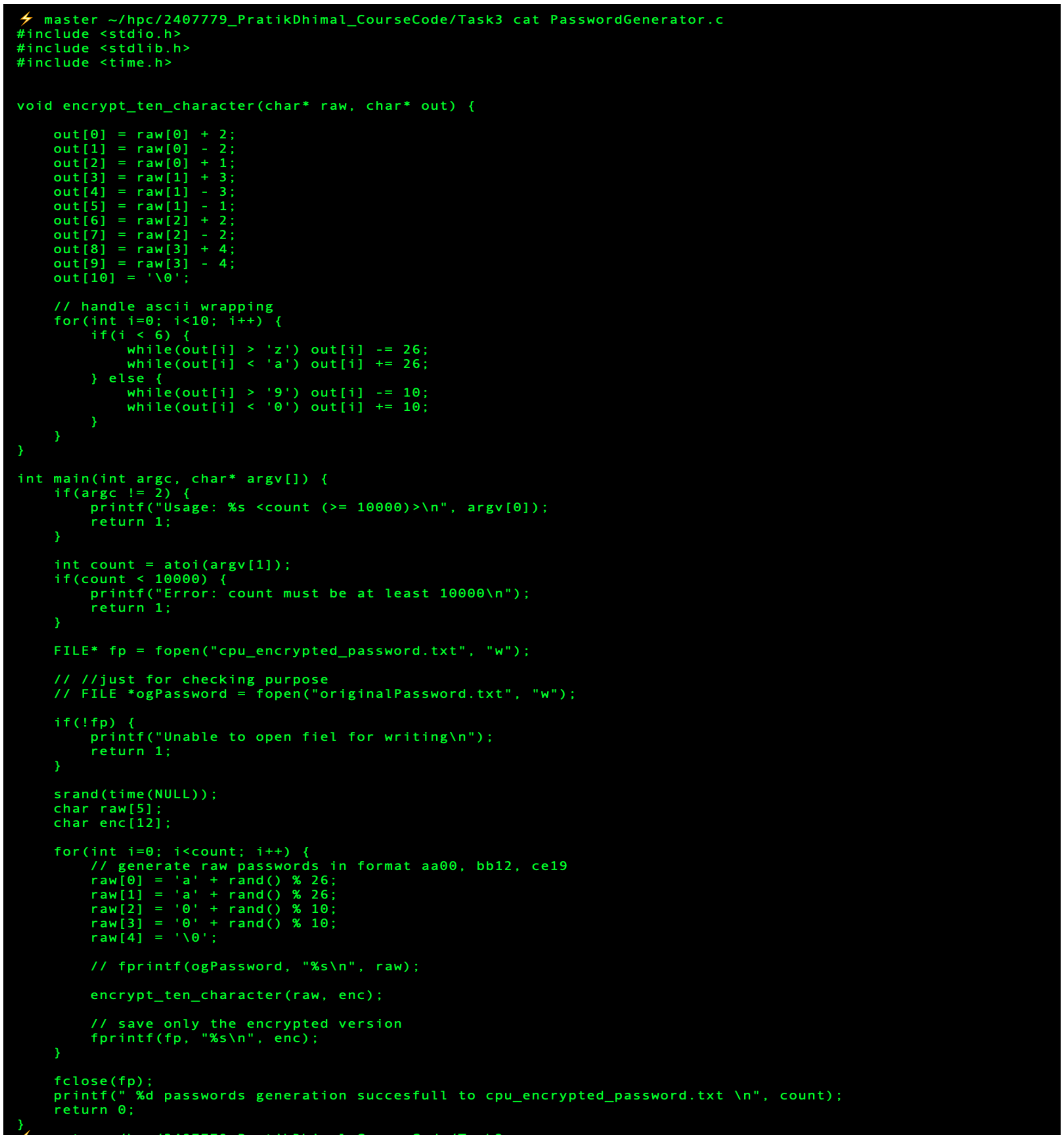
2.4. Transfer Edege detected value safely to the CPU

As shwon in **Figure 11** ,the **cudeMemcpy()** function is used with direction argument **cudaMemcpyDeviceToHost** to copy the final manipulated edege detected pixel value to the **h_out_rgba.** Now our Host(CPU) has the final sets of pixel value which could be easily encoded to a png image using the lodepng.

2.5. Making final image and memory deallocation

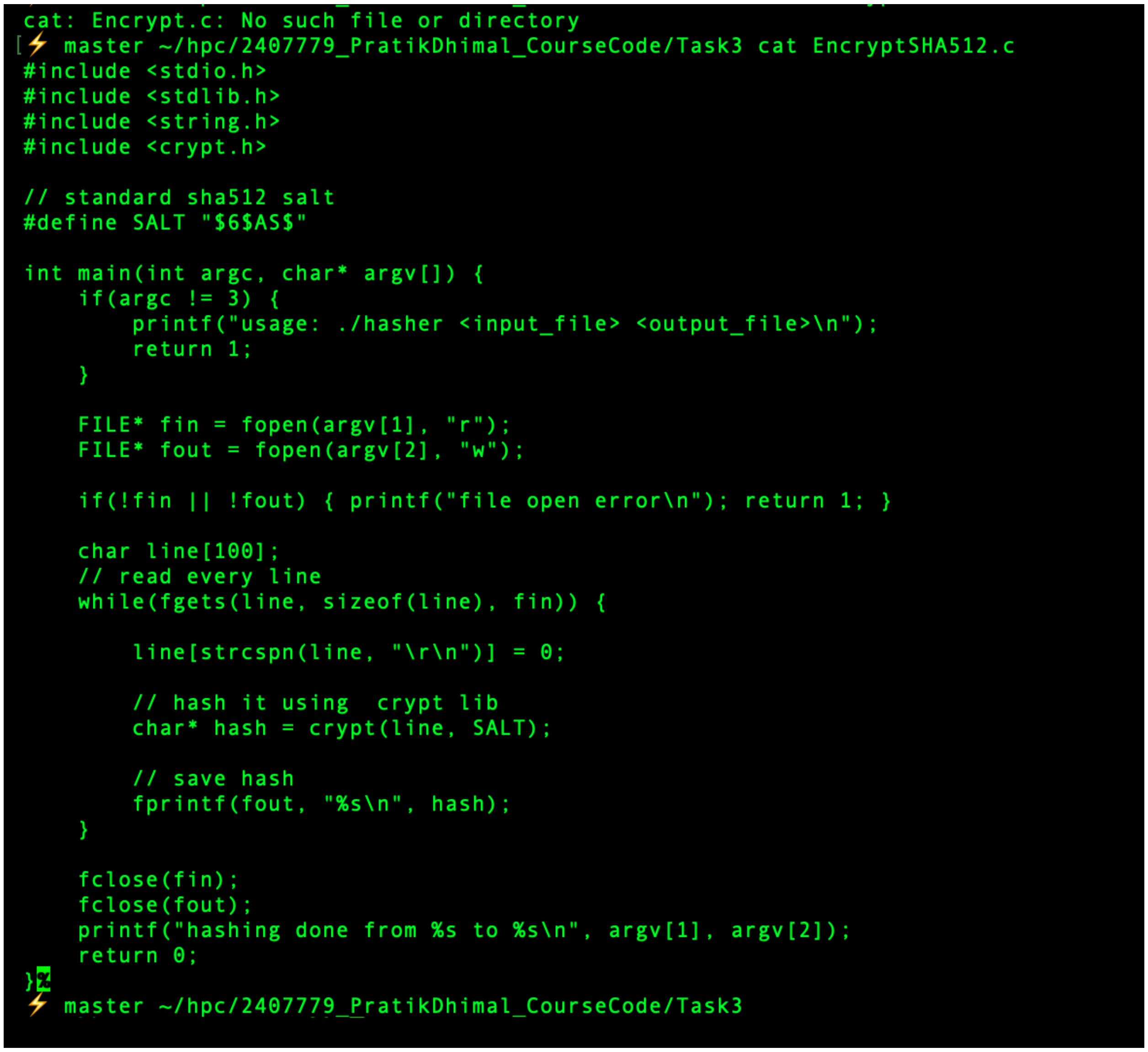
Finally the **lodepng_encode32_file()** function is used to encode the pixel back to png image. This gives us a final png image named **sobel-final-image.png** as shown in **Figure 14** .Lastly, we free all the memory used by the GPU as well as CPU to prevent memory leaking.

3. Appendix

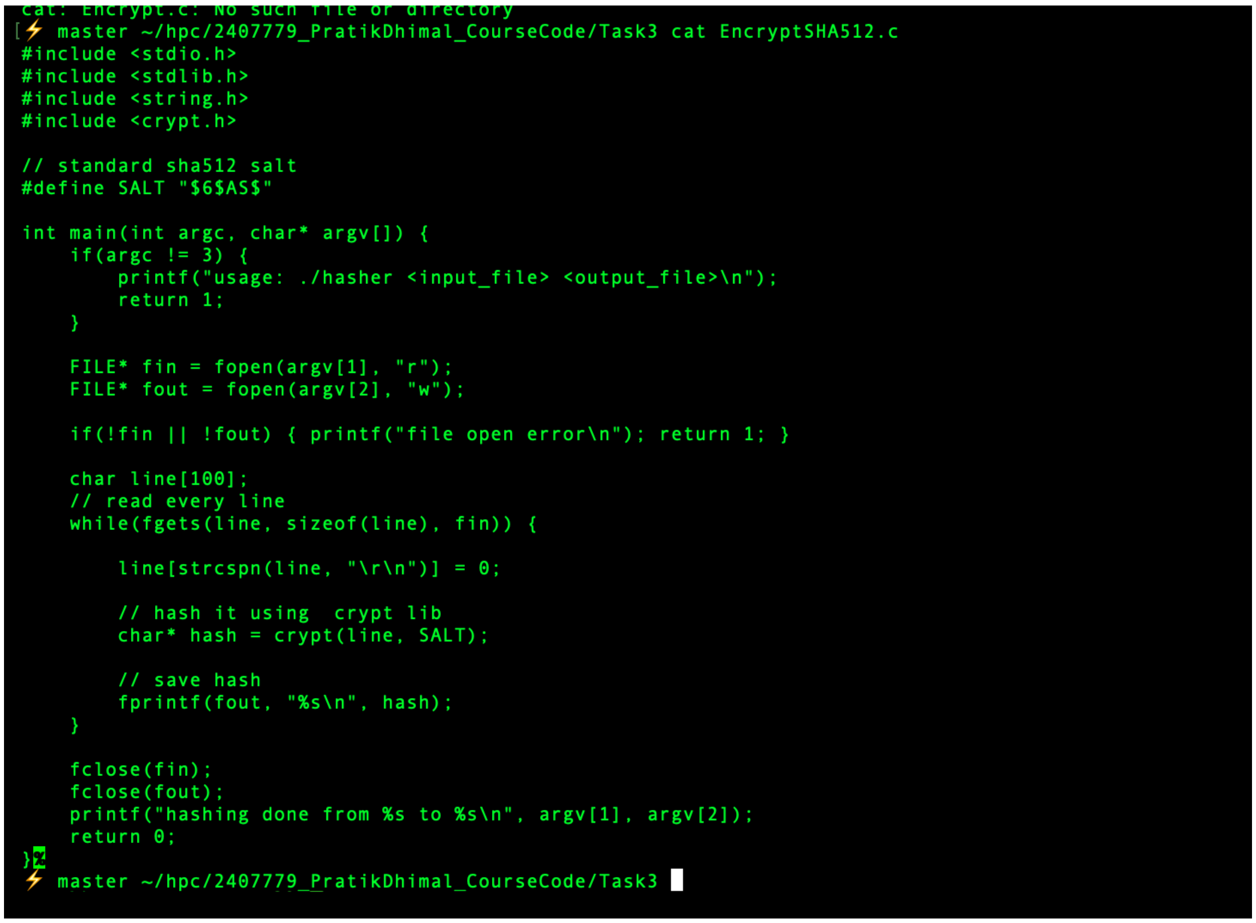


```
⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3 cat PasswordGenerator.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


void encrypt_ten_character(char* raw, char* out) {

    out[0] = raw[0] + 2;
    out[1] = raw[0] - 2;
    out[2] = raw[0] + 1;
    out[3] = raw[1] + 3;
    out[4] = raw[1] - 3;
    out[5] = raw[1] - 1;
    out[6] = raw[2] + 2;
    out[7] = raw[2] - 2;
    out[8] = raw[3] + 4;
    out[9] = raw[3] - 4;
    out[10] = '\0';

    // handle ascii wrapping
    for(int i=0; i<10; i++) {
        if(i < 6) {
            while(out[i] > 'z') out[i] -= 26;
            while(out[i] < 'a') out[i] += 26;
        } else {
            while(out[i] > '9') out[i] -= 10;
            while(out[i] < '0') out[i] += 10;
        }
    }
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        printf("Usage: %s <count (>= 10000)>\n", argv[0]);
        return 1;
    }

    int count = atoi(argv[1]);
    if(count < 10000) {
        printf("Error: count must be at least 10000\n");
        return 1;
    }

    FILE* fp = fopen("cpu_encrypted_password.txt", "w");

    // //just for checking purpose
    // FILE *ogPassword = fopen("originalPassword.txt", "w");

    if(!fp) {
        printf("Unable to open fiel for writing\n");
        return 1;
    }

    srand(time(NULL));
    char raw[5];
    char enc[12];

    for(int i=0; i<count; i++) {
        // generate raw passwords in format aa00, bb12, ce19
        raw[0] = 'a' + rand() % 26;
        raw[1] = 'a' + rand() % 26;
        raw[2] = '0' + rand() % 10;
        raw[3] = '0' + rand() % 10;
        raw[4] = '\0';

        // fprintf(ogPassword, "%s\n", raw);

        encrypt_ten_character(raw, enc);

        // save only the encrypted version
        fprintf(fp, "%s\n", enc);
    }

    fclose(fp);
    printf(" %d passwords generation succesfull to cpu_encrypted_password.txt \n", count);
    return 0;
}
```

Figure 1 PasswordGenerator.c

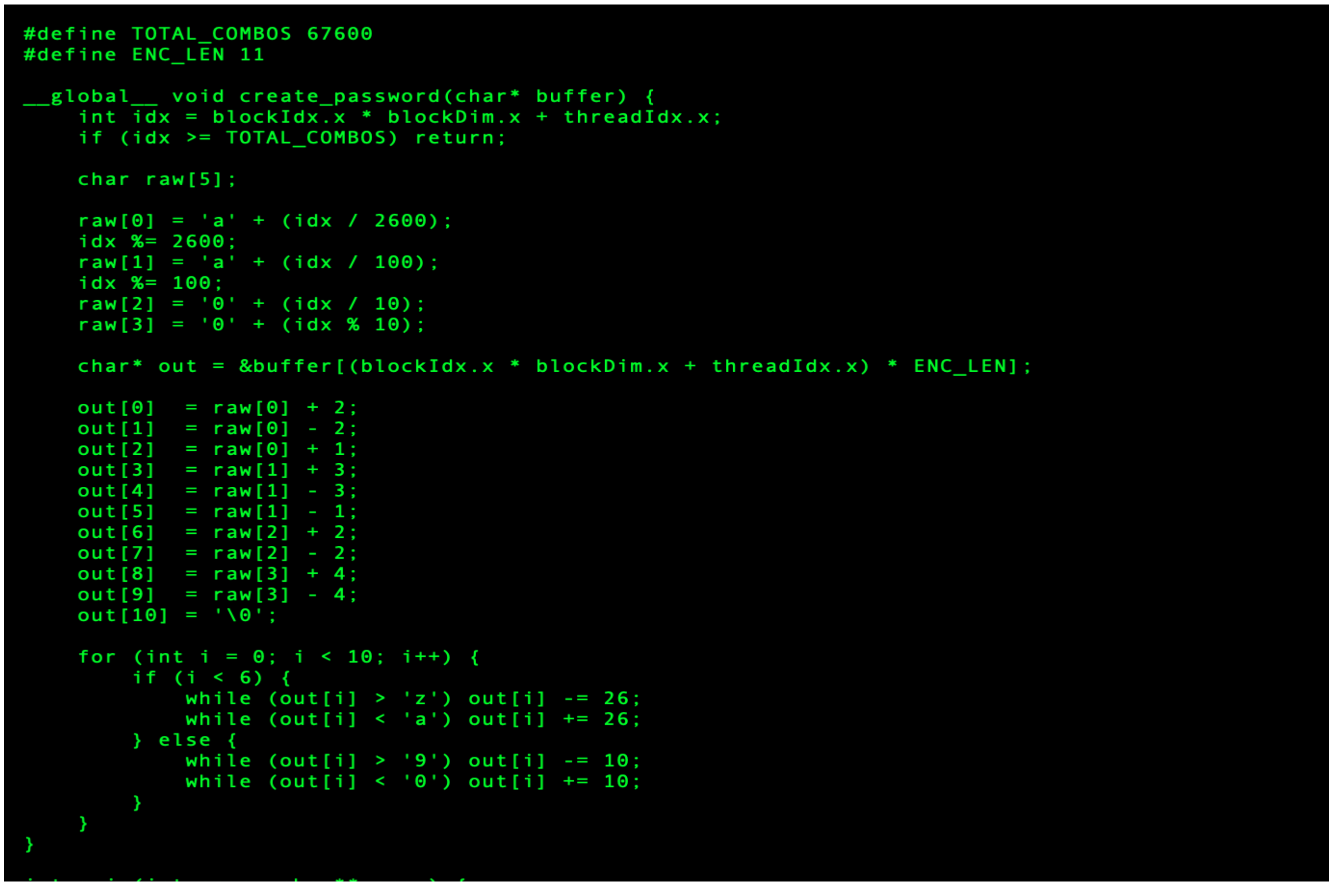


```
cat: Encrypt.c: No such file or directory
[⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3 cat EncryptSHA512.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crypt.h>

// standard sha512 salt
#define SALT "$6$AS$"

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf("usage: ./hasher <input_file> <output_file>\n");
        return 1;
    }

    FILE* fin = fopen(argv[1], "r");
    FILE* fout = fopen(argv[2], "w");

    if(!fin || !fout) { printf("file open error\n"); return 1; }

    char line[100];
    // read every line
    while(fgets(line, sizeof(line), fin)) {

        line[strcspn(line, "\r\n")] = 0;

        // hash it using  crypt lib
        char* hash = crypt(line, SALT);

        // save hash
        fprintf(fout, "%s\n", hash);
    }

    fclose(fin);
    fclose(fout);
    printf("hashing done from %s to %s\n", argv[1], argv[2]);
    return 0;
}%
⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3
```



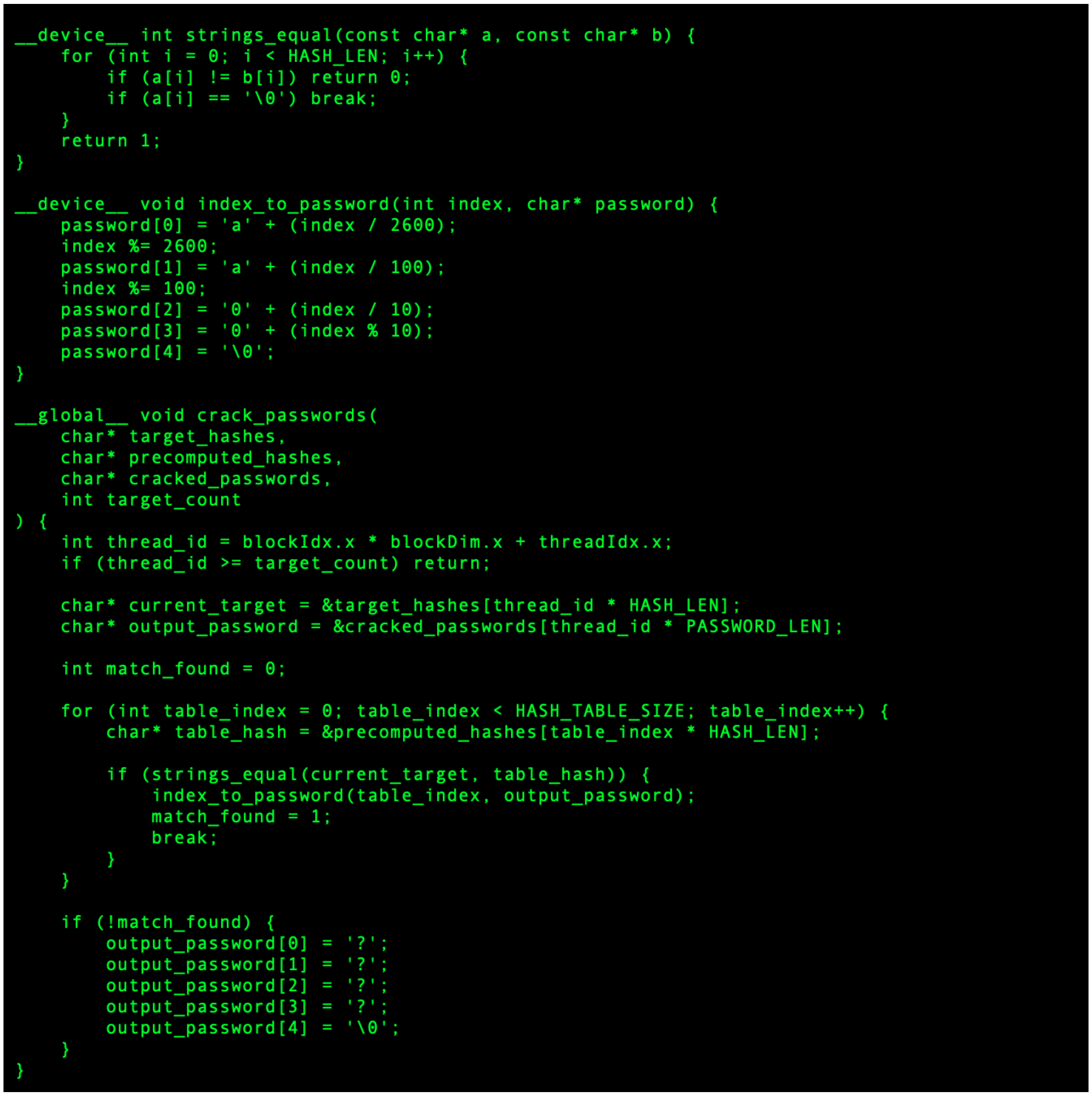
```
cat: Encrypt.c: No such file or directory
[⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3 cat EncryptSHA512.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crypt.h>

// standard sha512 salt
#define SALT "$6$AS$"

int main(int argc, char* argv[]) {
    if(argc != 3) {
        printf("usage: ./hasher <input_file> <output_file>\n");
        return 1;
    }

    FILE* fin = fopen(argv[1], "r");
    FILE* fout = fopen(argv[2], "w");

    if(!fin || !fout) { printf("file open error\n"); return 1; }

    char line[100];
    // read every line
    while(fgets(line, sizeof(line), fin)) {

        line[strcspn(line, "\r\n")] = 0;

        // hash it using  crypt lib
        char* hash = crypt(line, SALT);

        // save hash
        fprintf(fout, "%s\n", hash);
    }

    fclose(fin);
    fclose(fout);
    printf("hashing done from %s to %s\n", argv[1], argv[2]);
    return 0;
}%
⚡ master ~/hpc/2407779_PratikDhimal_CourseCode/Task3
```

Figure 2 EncryptSHA512.c



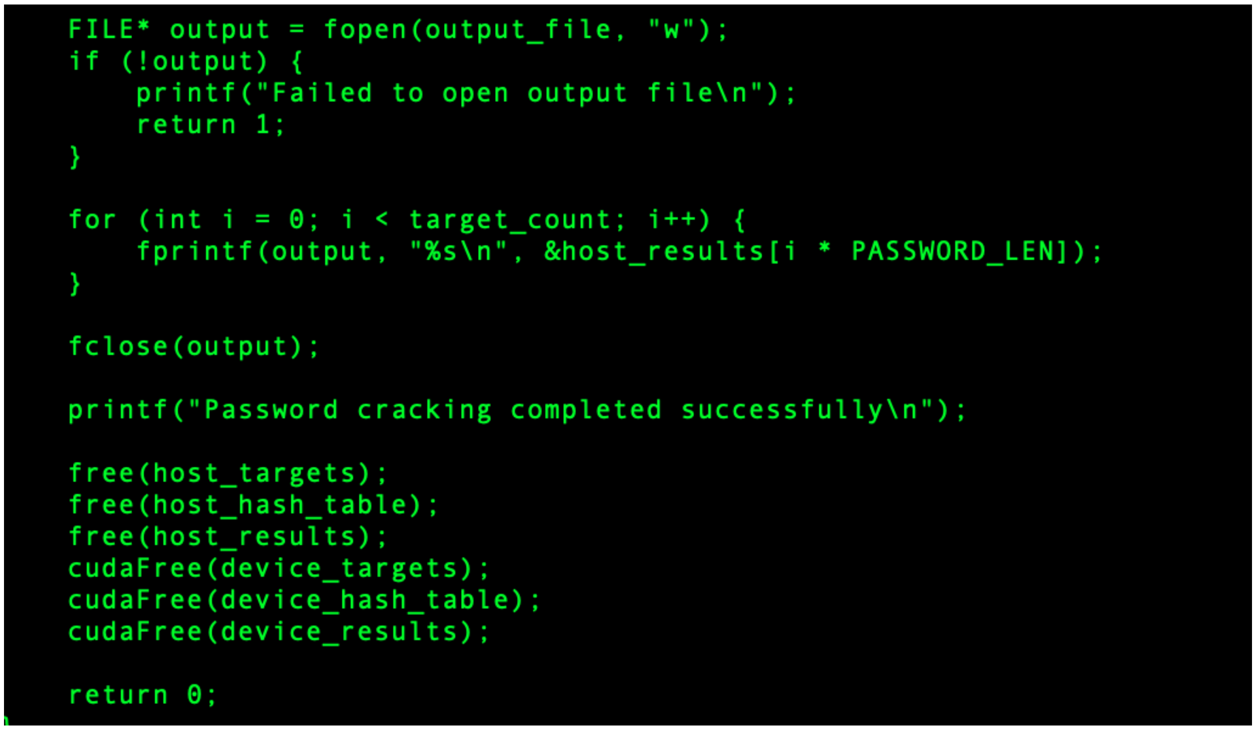
```
#define TOTAL_COMBOS 67600
#define ENC_LEN 11

__global__ void create_password(char* buffer) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= TOTAL_COMBOS) return;

    char raw[5];

    raw[0] = 'a' + (idx / 2600);
    idx %= 2600;
    raw[1] = 'a' + (idx / 100);
    idx %= 100;
    raw[2] = '0' + (idx / 10);
    raw[3] = '0' + (idx % 10);

    char* out = &buffer[(blockIdx.x * blockDim.x + threadIdx.x) * ENC_LEN];

    out[0]  = raw[0] + 2;
    out[1]  = raw[0] - 2;
    out[2]  = raw[0] + 1;
    out[3]  = raw[1] + 3;
    out[4]  = raw[1] - 3;
    out[5]  = raw[1] - 1;
    out[6]  = raw[2] + 2;
    out[7]  = raw[2] - 2;
    out[8]  = raw[3] + 4;
    out[9]  = raw[3] - 4;
    out[10] = '\0';

    for (int i = 0; i < 10; i++) {
        if (i < 6) {
            while (out[i] > 'z') out[i] -= 26;
            while (out[i] < 'a') out[i] += 26;
        } else {
            while (out[i] > '9') out[i] -= 10;
            while (out[i] < '0') out[i] += 10;
        }
    }
}
```

Figure 3 Password Generation Kernel



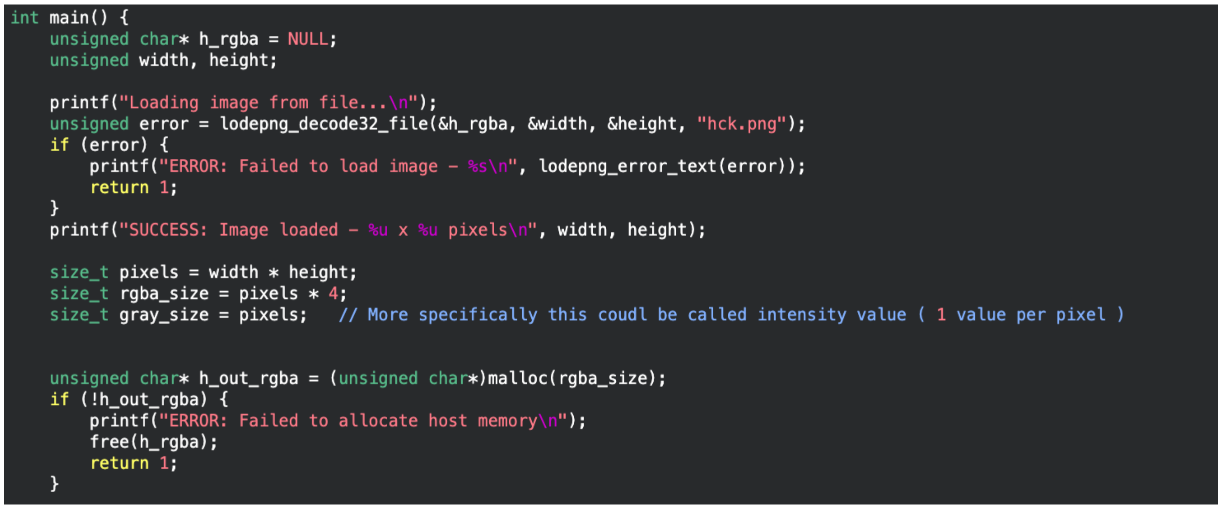
```
__device__ int strings_equal(const char* a, const char* b) {
    for (int i = 0; i < HASH_LEN; i++) {
        if (a[i] != b[i]) return 0;
        if (a[i] == '\0') break;
    }
    return 1;
}

__device__ void index_to_password(int index, char* password) {
    password[0] = 'a' + (index / 2600);
    index %= 2600;
    password[1] = 'a' + (index / 100);
    index %= 100;
    password[2] = '0' + (index / 10);
    password[3] = '0' + (index % 10);
    password[4] = '\0';
}

__global__ void crack_passwords(
    char* target_hashes,
    char* precomputed_hashes,
    char* cracked_passwords,
    int target_count
) {
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id >= target_count) return;

    char* current_target = &target_hashes[thread_id * HASH_LEN];
    char* output_password = &cracked_passwords[thread_id * PASSWORD_LEN];

    int match_found = 0;

    for (int table_index = 0; table_index < HASH_TABLE_SIZE; table_index++) {
        char* table_hash = &precomputed_hashes[table_index * HASH_LEN];

        if (strings_equal(current_target, table_hash)) {
            index_to_password(table_index, output_password);
            match_found = 1;
            break;
        }
    }

    if (!match_found) {
        output_password[0] = '?';
        output_password[1] = '?';
        output_password[2] = '?';
        output_password[3] = '?';
        output_password[4] = '\0';
    }
}
```

Figure 4 Crack Password kernels



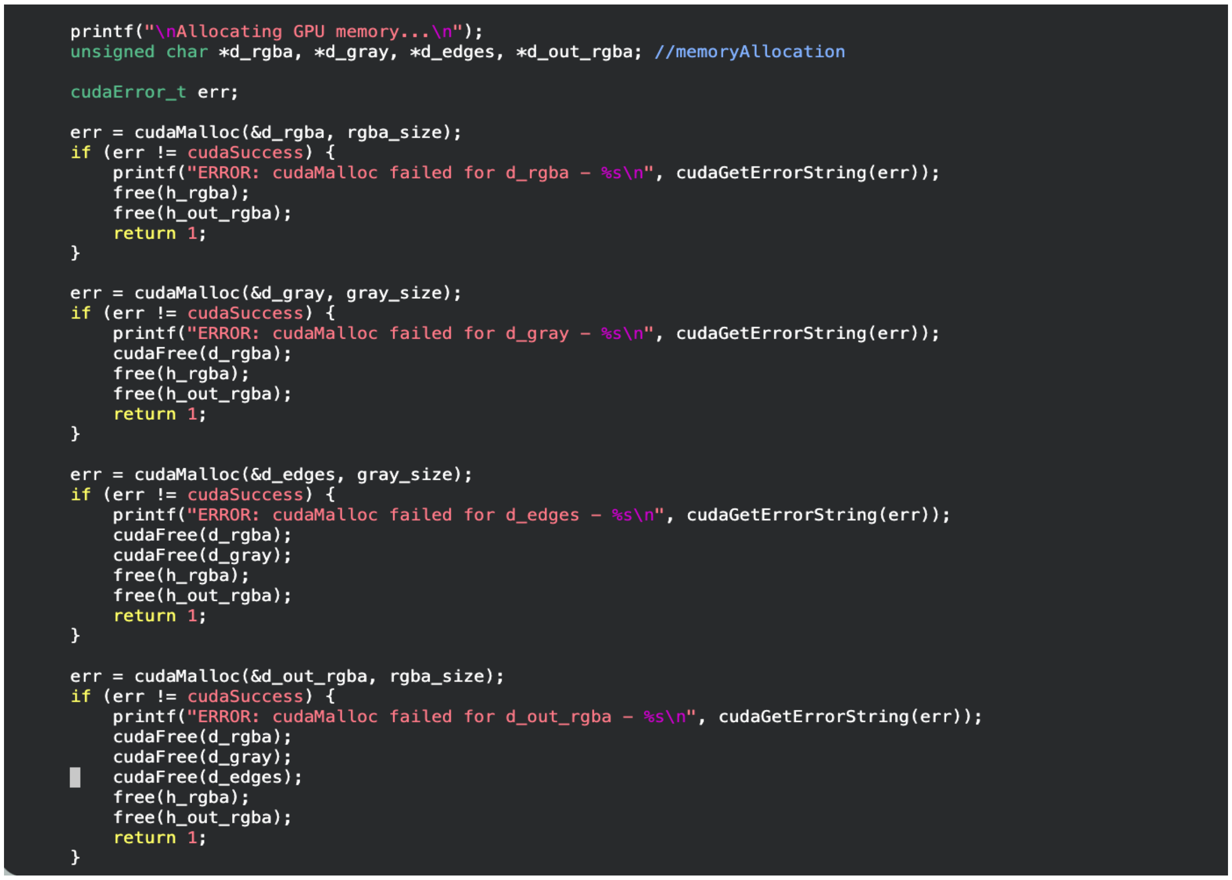
```
    FILE* output = fopen(output_file, "w");
    if (!output) {
        printf("Failed to open output file\n");
        return 1;
    }

    for (int i = 0; i < target_count; i++) {
        fprintf(output, "%s\n", &host_results[i * PASSWORD_LEN]);
    }

    fclose(output);

    printf("Password cracking completed successfully\n");

    free(host_targets);
    free(host_hash_table);
    free(host_results);
    cudaFree(device_targets);
    cudaFree(device_hash_table);
    cudaFree(device_results);

    return 0;
```

Figure 5 Memory Deallocation



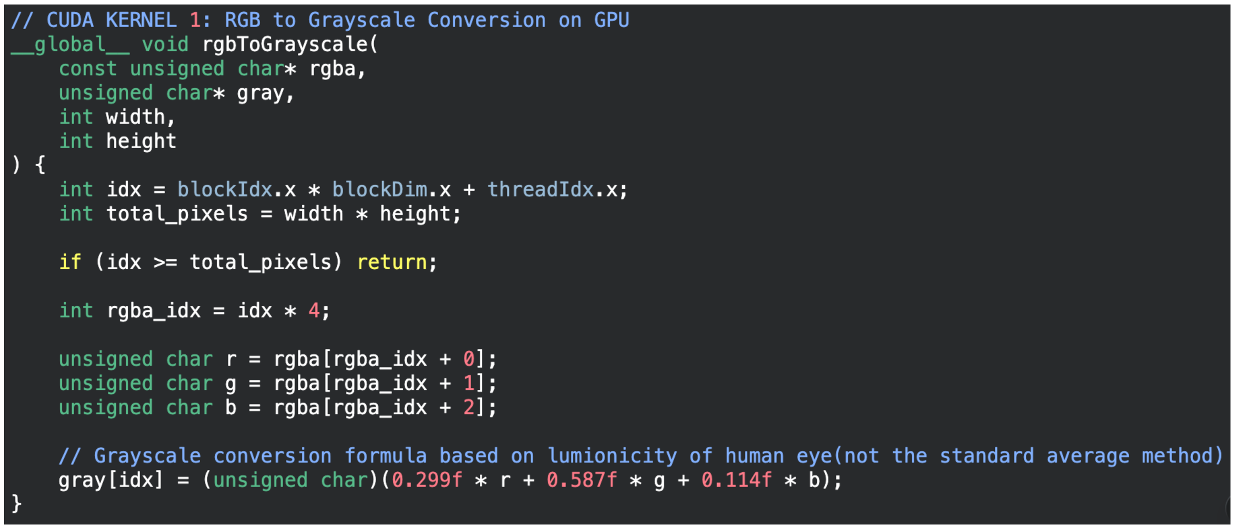
```c
int main() {
    unsigned char* h_rgba = NULL;
    unsigned width, height;

    printf("Loading image from file...\n");
    unsigned error = lodepng_decode32_file(&h_rgba, &width, &height, "hck.png");
    if (error) {
        printf("ERROR: Failed to load image - %s\n", lodepng_error_text(error));
        return 1;
    }
    printf("SUCCESS: Image loaded - %u x %u pixels\n", width, height);

    size_t pixels = width * height;
    size_t rgba_size = pixels * 4;
    size_t gray_size = pixels;   // More specifically this coudl be called intensity value ( 1 value per pixel )


    unsigned char* h_out_rgba = (unsigned char*)malloc(rgba_size);
    if (!h_out_rgba) {
        printf("ERROR: Failed to allocate host memory\n");
        free(h_rgba);
        return 1;
    }
```

Figure 6 loadign image to cpu memory



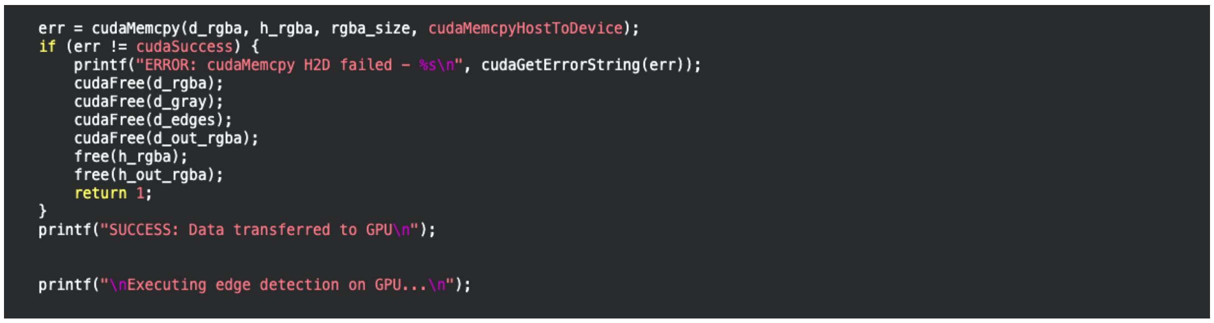
```c
    printf("\nAllocating GPU memory...\n");
    unsigned char *d_rgba, *d_gray, *d_edges, *d_out_rgba; //memoryAllocation

    cudaError_t err;

    err = cudaMalloc(&d_rgba, rgba_size);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMalloc failed for d_rgba - %s\n", cudaGetErrorString(err));
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }

    err = cudaMalloc(&d_gray, gray_size);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMalloc failed for d_gray - %s\n", cudaGetErrorString(err));
        cudaFree(d_rgba);
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }

    err = cudaMalloc(&d_edges, gray_size);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMalloc failed for d_edges - %s\n", cudaGetErrorString(err));
        cudaFree(d_rgba);
        cudaFree(d_gray);
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }

    err = cudaMalloc(&d_out_rgba, rgba_size);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMalloc failed for d_out_rgba - %s\n", cudaGetErrorString(err));
        cudaFree(d_rgba);
        cudaFree(d_gray);
        cudaFree(d_edges);
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }
```

Figure 7 Memory Allocation to GPU



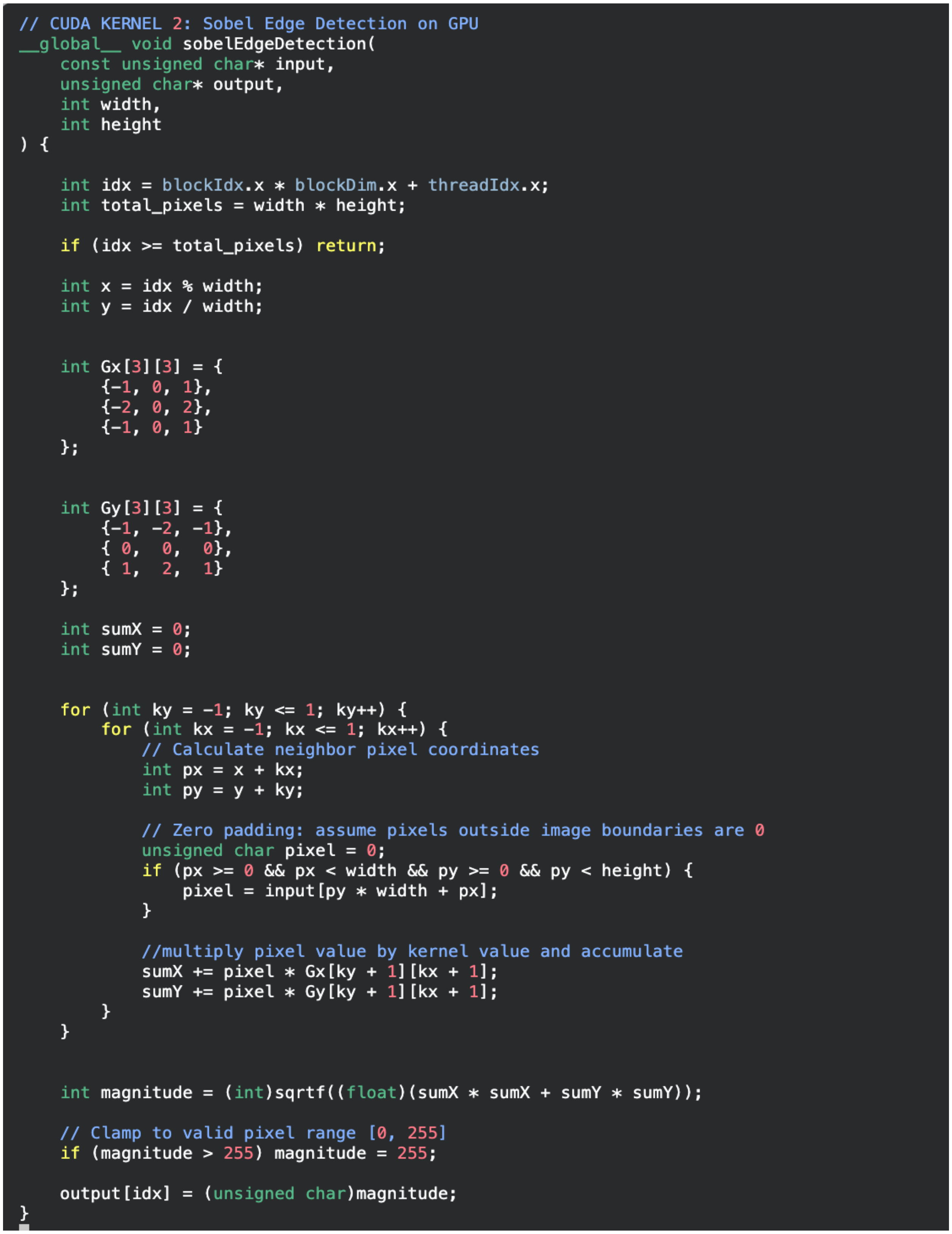
```
// CUDA KERNEL 1: RGB to Grayscale Conversion on GPU
__global__ void rgbToGrayscale(
    const unsigned char* rgba,
    unsigned char* gray,
    int width,
    int height
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    int rgba_idx = idx * 4;

    unsigned char r = rgba[rgba_idx + 0];
    unsigned char g = rgba[rgba_idx + 1];
    unsigned char b = rgba[rgba_idx + 2];

    // Grayscale conversion formula based on lumionicity of human eye(not the standard average method)
    gray[idx] = (unsigned char)(0.299f * r + 0.587f * g + 0.114f * b);
}
```

Figure 8 rgb to grayscale kernel



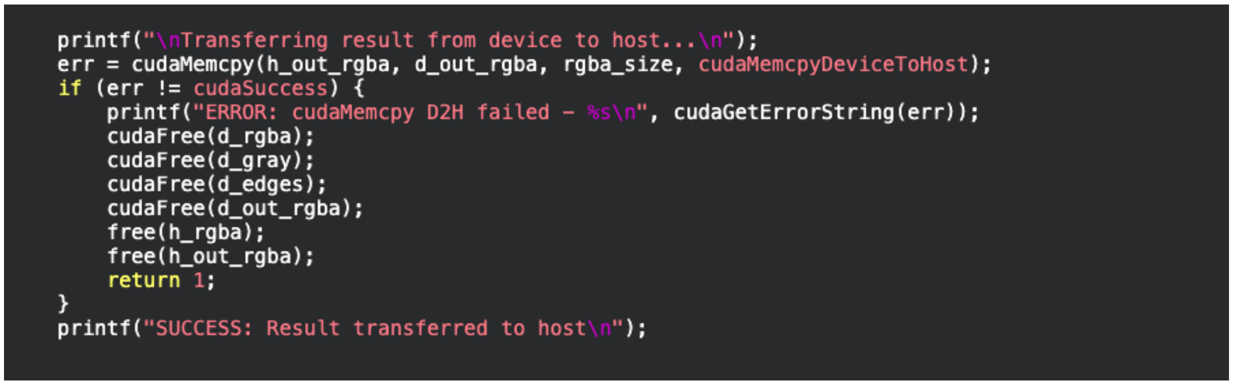
```
    err = cudaMemcpy(d_rgba, h_rgba, rgba_size, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMemcpy H2D failed - %s\n", cudaGetErrorString(err));
        cudaFree(d_rgba);
        cudaFree(d_gray);
        cudaFree(d_edges);
        cudaFree(d_out_rgba);
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }
    printf("SUCCESS: Data transferred to GPU\n");


    printf("\nExecuting edge detection on GPU...\n");
```

Figure 9 copying data to GPU for pexel manipulation



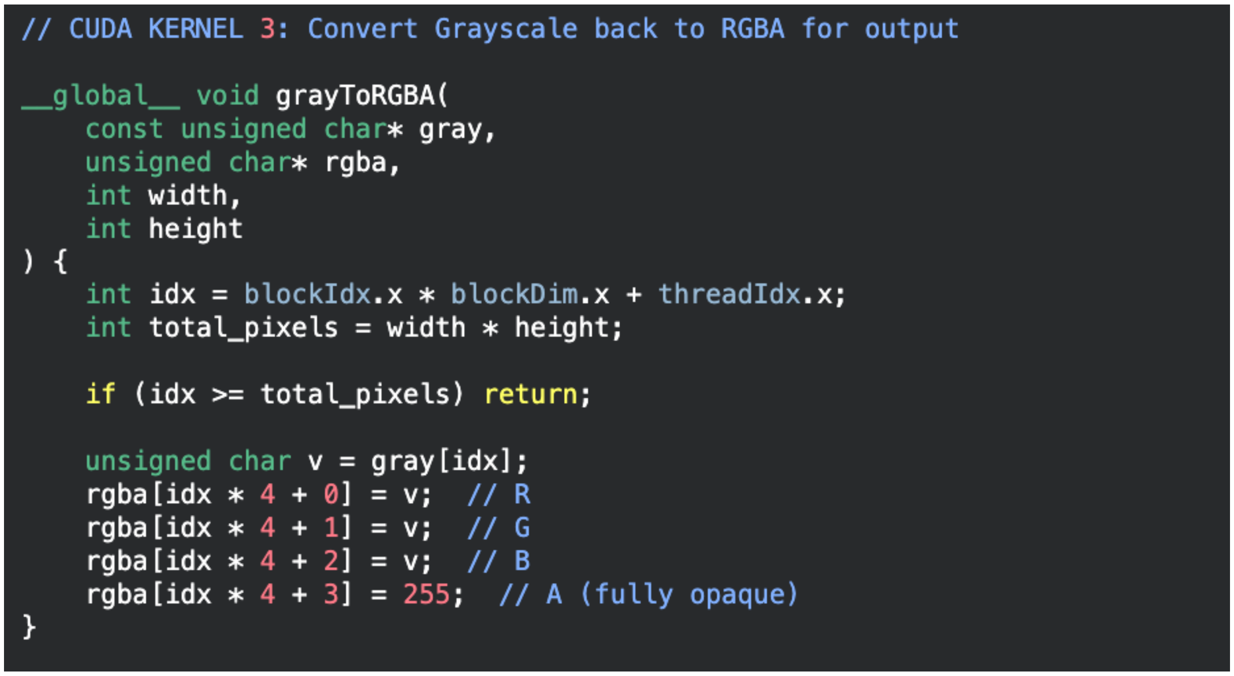
```
// CUDA KERNEL 2: Sobel Edge Detection on GPU
__global__ void sobelEdgeDetection(
    const unsigned char* input,
    unsigned char* output,
    int width,
    int height
) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    int x = idx % width;
    int y = idx / width;


    int Gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };


    int Gy[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    int sumX = 0;
    int sumY = 0;


    for (int ky = -1; ky <= 1; ky++) {
        for (int kx = -1; kx <= 1; kx++) {
            // Calculate neighbor pixel coordinates
            int px = x + kx;
            int py = y + ky;

            // Zero padding: assume pixels outside image boundaries are 0
            unsigned char pixel = 0;
            if (px >= 0 && px < width && py >= 0 && py < height) {
                pixel = input[py * width + px];
            }

            //multiply pixel value by kernel value and accumulate
            sumX += pixel * Gx[ky + 1][kx + 1];
            sumY += pixel * Gy[ky + 1][kx + 1];
        }
    }


    int magnitude = (int)sqrtf((float)(sumX * sumX + sumY * sumY));

    // Clamp to valid pixel range [0, 255]
    if (magnitude > 255) magnitude = 255;

    output[idx] = (unsigned char)magnitude;
}
```

Figure 10 sobel edege detection kernel



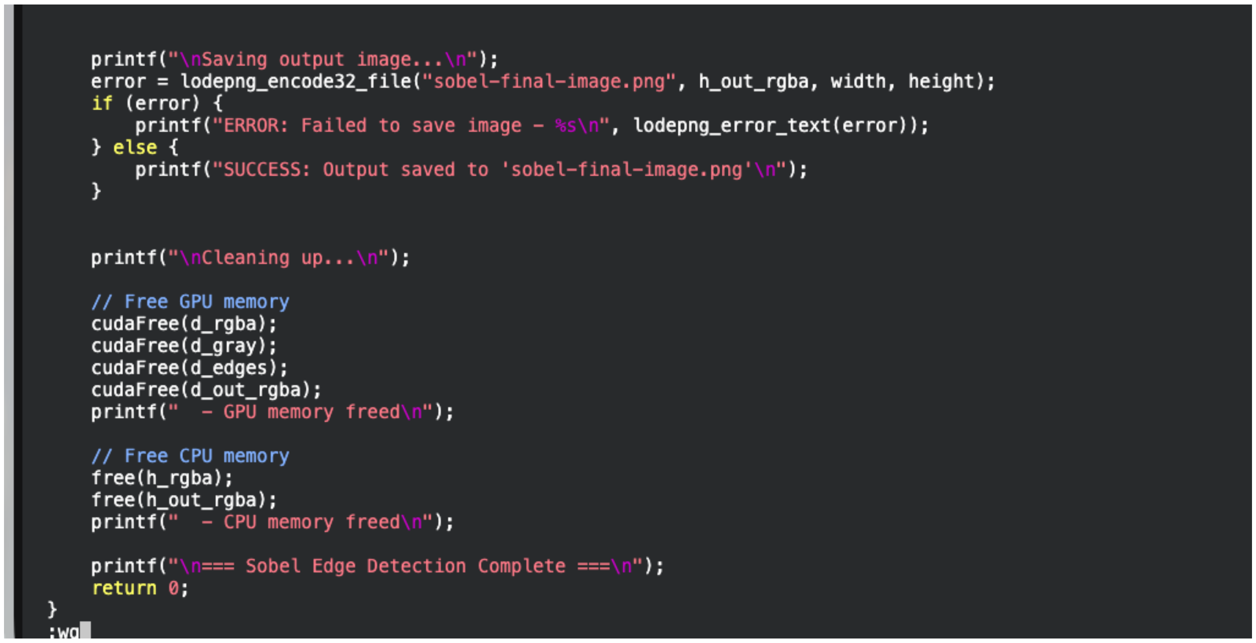
```
    printf("\nTransferring result from device to host...\n");
    err = cudaMemcpy(h_out_rgba, d_out_rgba, rgba_size, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        printf("ERROR: cudaMemcpy D2H failed - %s\n", cudaGetErrorString(err));
        cudaFree(d_rgba);
        cudaFree(d_gray);
        cudaFree(d_edges);
        cudaFree(d_out_rgba);
        free(h_rgba);
        free(h_out_rgba);
        return 1;
    }
    printf("SUCCESS: Result transferred to host\n");
```

Figure 11Final rgba value copy to host



```
// CUDA KERNEL 3: Convert Grayscale back to RGBA for output

__global__ void grayToRGBA(
    const unsigned char* gray,
    unsigned char* rgba,
    int width,
    int height
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_pixels = width * height;

    if (idx >= total_pixels) return;

    unsigned char v = gray[idx];
    rgba[idx * 4 + 0] = v;  // R
    rgba[idx * 4 + 1] = v;  // G
    rgba[idx * 4 + 2] = v;  // B
    rgba[idx * 4 + 3] = 255;  // A (fully opaque)
}
```

Figure 12 graysacel to rgba kernel



```
    printf("\nSaving output image...\n");
    error = lodepng_encode32_file("sobel-final-image.png", h_out_rgba, width, height);
    if (error) {
        printf("ERROR: Failed to save image - %s\n", lodepng_error_text(error));
    } else {
        printf("SUCCESS: Output saved to 'sobel-final-image.png'\n");
    }


    printf("\nCleaning up...\n");

    // Free GPU memory
    cudaFree(d_rgba);
    cudaFree(d_gray);
    cudaFree(d_edges);
    cudaFree(d_out_rgba);
    printf("  - GPU memory freed\n");

    // Free CPU memory
    free(h_rgba);
    free(h_out_rgba);
    printf("  - CPU memory freed\n");

    printf("\n=== Sobel Edge Detection Complete ===\n");
    return 0;
}
:wq
```

Figure 13 memory deallocation

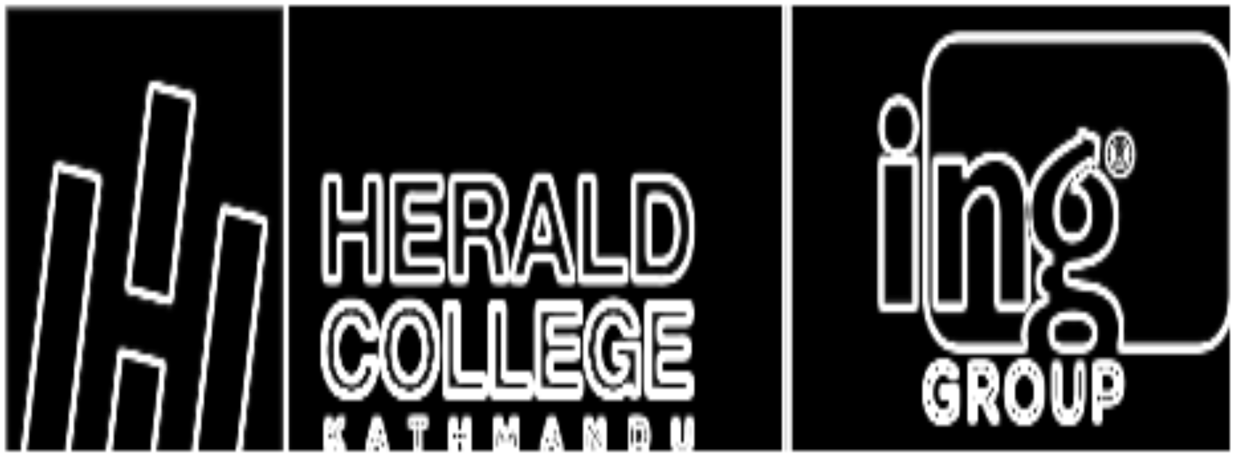
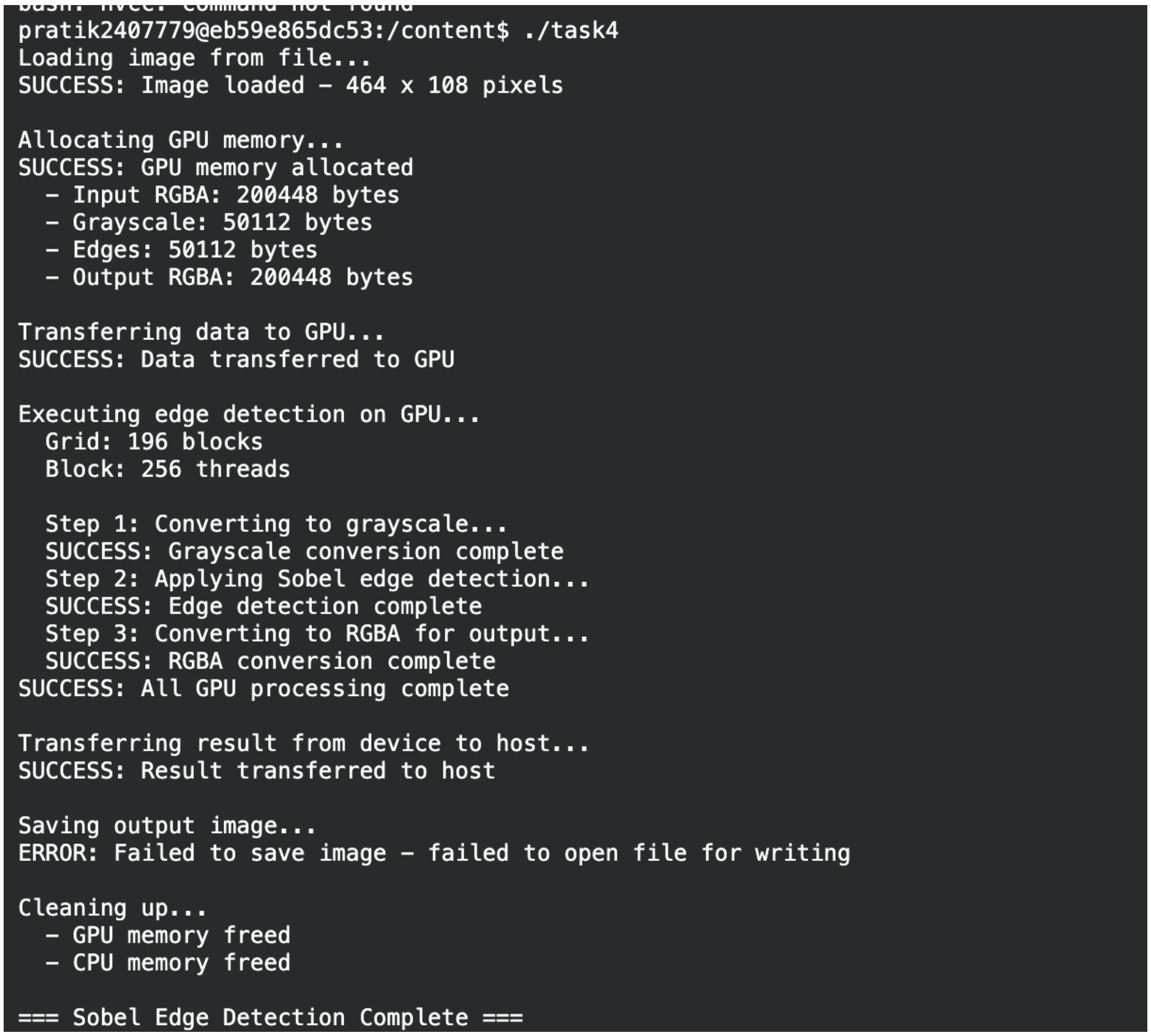


Figure 14 output image



```
bash: nvcc: command not found
pratik2407779@eb59e865dc53:/content$ ./task4
Loading image from file...
SUCCESS: Image loaded - 464 x 108 pixels

Allocating GPU memory...
SUCCESS: GPU memory allocated
  - Input RGBA: 200448 bytes
  - Grayscale: 50112 bytes
  - Edges: 50112 bytes
  - Output RGBA: 200448 bytes

Transferring data to GPU...
SUCCESS: Data transferred to GPU

Executing edge detection on GPU...
  Grid: 196 blocks
  Block: 256 threads

  Step 1: Converting to grayscale...
  SUCCESS: Grayscale conversion complete
  Step 2: Applying Sobel edge detection...
  SUCCESS: Edge detection complete
  Step 3: Converting to RGBA for output...
  SUCCESS: RGBA conversion complete
SUCCESS: All GPU processing complete

Transferring result from device to host...
SUCCESS: Result transferred to host

Saving output image...
ERROR: Failed to save image - failed to open file for writing

Cleaning up...
  - GPU memory freed
  - CPU memory freed

=== Sobel Edge Detection Complete ===
```

Figure 15 output commandline